ESD-TR-68-154

PROCEEDINGS OF PL/I SEMINARS, December 5, 1967 and March 5, 1968

April 1968

1 of 2
ESLFE

COMMAND SYSTEMS DIVISION
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

(Prepared under Contract No. F19628-67-C-0396 by Logicon, Incorporated, 255 West Fifth Street, San Pedro, California.)

LEGAL NOTICE

When U. S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

ESD-TR-68-154

PROCEEDINGS OF PL/I SEMINARS,
December 5, 1967 and March 5, 1968

April 1968

COMMAND SYSTEMS DIVISION
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

(Prepared under Contract No. F19628-67-C-0396 by Logicon, Incorporated, 255 West Fifth Street, San Pedro, California.)

FOREWORD

This report presents the proceedings of two seminars conducted by Logicon under contract F19628-67-C-0396 with the Electronic Systems Division of the Air Force Systems Command. The contract was sponsored under Project 6917, Task 8, by the Air Force Directorate of Data Automation. Mr. George Reynolds (ESLFE) was the Electronic Systems Division project monitor and Mr. William Rinehuls (AFADAB) was the cognizant task leader.

The overall purpose of the contract was to provide first-hand data on experience with PL/I to support subsequent evaluations of its usefulness in Air Force applications of the future. Two data categories were within the scope of the contract, the chief one being written presentation of information gained through the coding and checkout of benchmark problems in PL/I and another higher level programming language. The detailed data developed by Logicon for this category are given in ESD-TR-68-150. The second category consisted of oral presentations by experts who had actively participated in the design of the language or who are engaged in implementing systems using PL/I. The seminars at which these presentations were made were held in the Command Presentation Room, Laurence G. Hanscom Field, Bedford, Massachusetts, on December 5, 1967 and March 5, 1968.

Transcriptions of tape recordings made through the facilities of the Command Presentation Room and in some cases manuscripts furnished by the speakers served as the basis for the material presented here. Mr. William J. Stoner was responsible for arrangements for the seminars and coordination of these proceedings. Mrs. Laurel Bentley was responsible for their preparation. These proceedings were submitted in April 1968. The Logicon report number is CS-6819-R0106.

This technical report has been reviewed and is approved.

William F. Heisler

WILLIAM F. HEISLER, Colonel, USAF
Chief, Command Systems Division

ABSTRACT

This report consists of the proceedings of two seminars at which presentations were made by experts who had actively participated in the design of the PL/I language or who are engaged in implementing systems using PL/I. Each presentation was followed by a question-and-answer session in which the audience participated, and each seminar included an informal panel discussion among the authorities present.

CONTENTS

EXPERIENCE WITH TEACHING PL/I

Frederick P. Brooks, Jr.
Professor and Chairman,
Department of Information Science
University of North Carolina
Chapel Hill, North Carolina

First, let me begin with a bit of context. Three universities in North Carolina located within 30 miles of each other—two state universities, one private university—have a joint computing center known as Triangle Universities Computing Center. This is located out in the woods, equidistant from us, and is operated via communications from the campuses where we have smaller machines. Here at the big center we have a System/360 Model 75, with large core storage. It is a file-based system; there are only five tape drives on it. At Duke University and North Carolina State University there are System/360 Model 30s, and at the University of North Carolina at Chapel Hill we have a Model 40. These are connected to the big machine by Telpac A phone lines. The machine is out in the woods rather than being on one of the campuses for reasons which are essentially political—very important. The three universities are equal partners in this center and run it as a separate nonprofit corporation. So I will be discussing the experience of the whole complex of three universities, with different ownerships and managements. These three essentially constitute one computer-using community, and I polled all over it to find out reactions and experiences in the use of PL/I.

At Chapel Hill, we began teaching PL/I in the fall of 1965, with a course in programming systems, an advanced course for students who already knew how to program computers. Now, this was about all we could do at that time because we had no compiler. In the spring of 1966 we began using the first prereleased compiler with the spring term advanced programming systems class. In the summer we began acquiring really substantial faculty experience in its use. In the fall of 1966, we switched all of our beginning programming classes to PL/I. Last fall we taught about 250 students; last spring we taught another crop of 280; and this fall we have a third crop of about 270 going through their initial introduction to computers and PL/I. At North Carolina State University in Raleigh and at Duke University they have been offering some instruction in PL/I and some in FORTRAN. As a community we have now a fair amount of experience. In January of this past year we switched from Release 1 to Release 2 of PL/I, which was a substantial improvement in many ways, and we are in the process of installing the third release now, which promises to make improvements in the efficiency of the object code.

What kinds of things do we do, and why, and how? First, consider the beginning programming courses using PL/I. At Duke they had 150 students

last year in the beginning programming course; at State they taught 60 chemical engineering students; we taught, as I said, about 530 students last year and we are teaching another 270 this fall. Our experience at Chapel Hill illustrates one of the things that I like most about PL/I. The large state-supported engineering school is at Raleigh at North Carolina State University; Chapel Hill is a liberal arts university with a medical complex, various professional schools, pure sciences, mathematics, statistics, social sciences. The demand for students to learn to use computers arises in all of these areas. Our beginning programming course is a high-level-language course; we teach machine language in the second term of a computer sequence. I think this is the right way to do it. I started off convinced that one should teach machine language first, then teach a high-level programming language. I have completely switched.

We have the following pair of problems: one, our students come from many backgrounds and have many interests in what one does with a computer; two, many of them are going to take further courses in computing and we want these to rest on a common foundation course. Our clientele falls into three general categories: 1) the science and math types, who want to learn principally about scientific computing; 2) the business types, who are especially interested in business data processing and in management decision-making; and 3) the natural-language types: students in English, social sciences, the Romance languages, journalism, all of whom are especially interested in using a computer for the purpose of taking English language text and performing information retrieval, syntax analysis, linguistic studies, etc.

We would like all these students to become proficient to a common level on which stands the rest of the curriculum. We therefore offer our beginning programming course in three flavors; we call these A, B and C. Each of these courses teaches PL/I. Each teaches the topics in different order. Each draws the examples from different subject matter, according to what the students are interested in. Each designs the student exercises according to the field of interest. In short, we tailor-make the student's introduction to computers according to the field of application in which he's interested.

In business data processing they start with input/output, for example. In the natural-language sections, they start with character strings and work with nothing but character strings for some time. By the end of the term, all sections are competent in a common language base. Other courses build on that. Now, this common-language/multiple-flavor approach would not be possible for us with any other high-level language, and it is one of the big advantages of PL/I. In such a class, the student works five to seven problems a semester, and our students take about five machine runs for each

problem. Let me read several remarks from the instructors about the kinds of exercises. From one of the science-math sections:

> "We ran five problems plus a term project. Problems one and two were based on polynomial evaluations; problem three was a Newton-Raphson method which finds roots of algebraic equations; problem four was a two-dimensional Monte Carlo integration; problem five was a simple alphanumeric search and count. Term projects included bidding bridge hands, coding and decoding messages, determining legal moves for checkers or chess, etc."

From a professor of chemical engineering at North Carolina State, who has been teaching sophomores:

> "I have been using PL/I extensively in the past year for my own research, and I have been teaching the PL/I language to our chemical engineering sophomores. The research involves the numerical solution of sets of partial differential equations in the general area of experimental data reduction and analysis. For this kind of work the array statements are especially useful. I am presently teaching PL/I to 60 chemical engineering sophomores. We are teaching PL/I in one class per week. We have given, so far, six lectures on the PL/I language, and the students have written and run six programs. I think the PL/I language is much easier to teach than any of the other languages I have used. Students in this course have been very successful in getting their programs to run, and I would estimate that 95% of them have turned in essentially correct programs on time. I would be very discouraged if I had to go back to teaching FORTRAN."

Now, let me say that this is a very skilled teacher, and he finds PL/I easier to teach because he's carefully selected the order in which he presents the topics. If I take a graduate student with no teaching experience and fling a PL/I manual at him and say, "Teach that to beginners," a disaster ensues.

The second area that I want to discuss is the use of PL/I as a tool for teaching advanced courses. Obviously, Numerical Analysis, a mathematics course, is one in which you need a programming language for the student to work his exercises; we use PL/I and find it quite satisfactory. The same is true in Mathematical Physics, an elementary theoretical methods course.

The same is true in our Business Data Processing course. This is an advanced course in which the students do very little programming; they are principally concerned with systems analysis, file organization, input verification, and the problems of setting up a business system. For what programming they do, we use PL/I as the tool and find it satisfactory. Then we have several courses in natural-language processing: one in syntax analyzers, one in information retrieval, and a general natural-language course that covers indexing and concordances and other topics. In all of these, then, we find it convenient to have the students already know a language which they can use.

Now, a special kind of course is Programming Systems. We do teach students how to build programming systems, and in this course we've undertaken some fairly elaborate laboratory projects. I divide the students into teams of six. In the first lab meeting of the course I explain that we're going to build something, and spend minutes describing the ultimate objective. Then I walk to the back of the room, and leave them, totally unstructured, to choose a team leader, decide on a plan of action, establish a schedule, and get going. They are almost invariably late, but that's part of learning how to build programming systems.

Some of the projects might be of interest to you. These have been built using PL/I as the tool, partly because there wasn't schedule time to use assembly language, and partly because PL/I is much easier to debug in. Last term, for example, one team built a thing called "PUNT," which was a Teletype-based PL/I syntax analyzer. One sits at the Teletype and keys in a statement. At the end of each statement the computer returns, statement by statement, diagnostics of syntax errors. It does not attempt to execute the program; it only looks at each statement to see if the programmer left out commas or the semicolon, used the wrong kind of function, or things like that. They built this, and got it running, and I was rather proud of them. It was a nice piece of work.

Another team built a system called Report Program Language (RPL). A report program generator is rather a convenient way of doing input and output: one specifies input by laying out cards; one specifies output by laying out printout sheets. The IBM report program generator has, however, an awkward language for what one does between input and output—their arithmetic specifications. So the team built a language that has RPG input and output facilities and has the full range of PL/I in the middle. They did this by using the macro facility in PL/I to translate the RPG input and output statements into PL/I input and output statements. It's not quite running, but they got well along.

One student text editor uses the graphical display. The Journalism Department receives the Associated Press wire service, which is entered into the computer on paper tape. The text editor displays the text from the AP wire on the 'scope. Then he takes the light pen, and strikes out, deletes, inserts, adds or edits; when he finishes, the system instantaneously right-justifies the text. The editor pushes the button, and the stuff is printed out, upper and lower case, on the high-speed printer downstairs. And, once again, this is the kind of program in which PL/I's facilities for character manipulation, tied with the ability to interact with the 'scope, are very important to us.

Another student has taken for his master's thesis the problem of writing a compiler for Iverson's language, APL. His assignment was to write a translating compiler with the least possible effort. He did this by using macro facility in PL/I statements, then harnessing the PL/I compiler to compile them. He uses a subset of Iverson's language, needless to say. He got a compiler in about six weeks of effort. I thought that was rather nice. It illustrates the power of the compile-time facility in PL/I.

Another student has been writing course material for a computer-assisted-instruction program. We have a little pilot project in computer-assisted instruction. We're trying to learn how one does this, and whether it really helps. One student has been preparing the control program in PL/I. We hope to put our first victims on beginning in the February semester.

Over at Duke in the Physics Department they've been using PL/I principally to write data reduction programs. They get volumes of data, pictures they take of bubble chamber events. They go to Brookhaven to turn on the big accelerator. They take perhaps 10,000 pictures, one every 2 seconds while they are on, and then they come trucking home to spend the next few months figuring out whether they caught anything or not. That's fishing with a fine net in a big ocean. They scan the film semiautomatically, getting the data into a computer, and then they compute for all those tracks, first finding out where the track is, then screening the interesting tracks from the uninteresting tracks.

Professor Ferrell in Chemical Engineering at State has been doing heat transfer and partial differential equations, data reduction and analysis. We have one man in Biochemistry who has been working on the structure of the protein molecule, essentially a geometric problem; he's been working with hemoglobin. Now, hemoglobin is an interesting molecule—apparently hemoglobin does not combine with oxygen in the lungs and carry the oxygen to the muscles. If it did, the undoing of the chemical bond between the hemoglobin and the oxygen would use

up a lot of energy in the muscles. What apparently happens is this: The hemoglobin has a basket shape in three dimensions, and when oxygen is dissolved in the blood in the lungs, it changes the acidity slightly. Under this condition the hemoglobin changes in shape and wraps around the oxygen molecules, and they are carried out to the muscles. Out at the muscles, the doing of work is generating lactic acid. That changes the acidity, so the hemoglobin opens up and makes that oxygen available to the muscles. It is difficult to conceive of a lower energy mechanism; I find it impossible to believe that all that mechanism didn't have a designer. Our biochemist has been working to establish the structure of hemoglobin. This is a very big combinatorial problem. They have been working in PL/I, and they are anxiously waiting the third release, since object-code performance is vital.

I wrote a little system this summer. We didn't have a good concordance-generating program. The technique is straightforward, so I decided I could spend an hour a day on it. The system starts from a Dura-paper-tape typewriter, which types in upper and lower case and punches paper tape. This goes to magnetic tape. Then a program goes through the text determining the word boundaries. Then it looks them up in a table of function words and throws out the the's and and's and some 200 other function words. It performs, in the process, a code conversion, from whatever code it was originally in to standard internal code. The entries are then sorted and printed in concordance form, with the word, a hundred characters of context, and page number.

It might be interesting for you to look at some PL/I, so I brought along two of the subroutines for this program. One of these is the program which does the code conversion; this is a byte-by-byte table look-up and translation. The second one is the one that does the search in the function word table. This is an ordinary binary search; it starts in the middle and goes up or down until it homes in on the word—returns a 1 if the word is in the list and a 0 if it is not. The procedure is straightforward. It illustrates an implementation dependency; it depends upon the use of 8 bits to represent the character. It also illustrates the uses of bit and character string manipulations.

Let's go on to the second slide. FNWORD is the procedure. It returns a 1 if a 20-character word is in the 256-word list. The words must be in alphabetical order. Look at the names of the variables; I have declared K, then, to be a character string 20 characters long. X is the table of words searched for the presence of K. It is 256 entries long, and each entry is 20 characters long. J, P and Q are fixed binary; they are, in fact, integers and are used as indices. The algorithm is straightforward, and I won't go through it. Both of these are

```
1     RECODE: PROC (S,T);                  /*A FUNCTION PROC TO TAKE S, A */
                                           /*VARYING-LENGTH CHARACTER STR */
                                           /*AND CODE-CUNVERT IT UISING   */
                                           /*A 256-BYTE ARRAY T.  THE PROC*/
                                           /*USES NO I/O AND THE METHOD   */
                                           /*IS EXACTLY THE SAME AS ASSY T*/
                                           /*TR. K IS RECODED IN PLACE, H */
                                           /*HENCE NOTHING ELSE IS RETURNE*/
                                           /*RETURNED.                    */
2       DCL  S CHAR (*) VARYING,           /*STRING TO BE RECODED         */
             T (0:255) BIT (8) PACKED,     /*RECODE TABLE AS BIT STRING   */
             V CHAR (1) STATIC,            /*TEMPORARY CHAR VARIABLE      */
            (I,                            /*INDEX OF CHAR BEING CODED    */
             IMAX,                         /*MAX VALUE OF I               */
             J) FIXED BIN STATIC;          /*INDEX IN RECODE TABLE        */
3       IMAX=LENGTH(S);                    /*                             */
4       DO I=1 TO IMAX;                    /*                             */
5         J=UNSPEC(SUBSTR(S,I,1));         /*USE CHAR AS INDEX            */
6         UNSPEC (V)= T(J);                /*PICK UP CHAR EQUIVALENT      */
7         SUBSTR (S,I,1)=V;                /*                             */
8       END;                               /*                             */
9       RETURN;                            /*                             */
10      END RECODE;                        /*                             */
```


Slide 1

```
1     FNWORD: PROC (K,X) BIT (1);          /*RETURNS A '1' IF A 20-CHAR   */
                                           /*WORD IS IN A 256-WORD LIST   */
                                           /*WORDS MUST BE IN ALPHABETIC  */
                                           /*AND THE HIGH-ORDER PART OF   */
                                           /*THE LIST MUST BE PADDED WITH */
                                           /*NINES                        */
2       DCL  K CHAR (20),                  /*WORD TO BE SEARCHED FOR      */
             X (256) CHAR (20),            /*FNWORD TABLE TO BE SCANNED   */
            (J,                            /*LIST SCAN INDEX              */
             P,                            /*LOWER SEARCH BOUND           */
             Q) FIXED BIN STATIC;          /*UPPER SEARCH BOUND           */
3     K1: P=1;                             /*                             */
4     K2: Q=256 + 1;                       /*256 IS DIM OF TABLE          */
5     K3: J=(P+Q)/2                        /*DEPENDS ON EVAL GIVING FLOOR */
6     K4: IF K=X(J) THEN RETURN ('1'B);    /*                             */
8           IF K>X(J) THEN GO TO K7;       /*                             */
10    K5: IF Q=J THEN RETURN ('0'B);       /*                             */
12    K6: Q=J;                             /*                             */
13        GO TO K3;                        /*                             */
14    K7: IF P=J THEN RETURN ('0'B);       /*                             */
16    K8: P=J;                             /*                             */
17        GO TO K3;                        /*                             */
18        END FNWORD;                      /*                             */
```


Slide 2

real working subroutines—but they are small and simple enough that I thought that you might be interested in looking at them.

I teach students to declare all variables. I think this is sound practice in any programming language that has declaration facilities. PL/I will allow one to do a lot of things by default. It also makes up things for you that are very interesting and occasionally amusing when you've defaulted on your options. I think it is a sound practice to declare all variables.

It is important in this language to use mixed numbers in fixed point only when you really want mixed numbers and really need fixed point. The fixed point is intended as an integer mode. It was generalized, principally to provide for the command and control application, so that a person can write highly efficient machine code in which the scaling is done for him. That's the only purpose for which fixed-point mixed numbers should be used. By mixed numbers I mean integers plus fractional parts. They must be used carefully, as carefully as though they were being scaled by hand. If the fixed point is used as an integer mode, just as in ALGOL or FORTRAN, it behaves the same way as the integer mode in those languages.

So much for the research area. The last area that you might find interesting is the North Carolina Computer Orientation Project. We have a scheme sponsored by the State Board of Higher Education to try to bring the blessings of computation to the far-flung corners of the state. In North Carolina that's pretty far, from the outer banks to way back in the mountains. This takes advantage of the new technical ability to operate computers from terminals. Under the Computer Orientation Project, we go to every post-high-school institution in the state and say, "If you want to, you may try our computer service for a year free. We furnish you a Teletype and communications. We furnish you enough machine time to run 100 jobs a month. Three circuit riders, who spend much of their time on the road, will help to answer your questions, teach your faculty, help you organize courses, etc." There are 86 eligible institutions in the state; 16 of them are on line today, and we are adding about two a month. The first one went on last April, so we've been adding two and three a month since April.

The users run the whole range from experienced faculty to beginning programmers. There are also beginning programmers among the faculty, and experienced people among the students. They are operating in batch mode from Teletype. They prepare the program on paper tape, transmit it, and get the answer back when the machine runs it, which may be five minutes to several hours on these problems.

EXPERIENCE WITH TEACHING PL/I

For use with a terminal, free-form language is essential. The most awkward part of our remote operation is the Job Control Language, which is not free-form. In fact, we have written a substitute job control language that a little editor reads from Teletype and translates and expands into regular JCL.

So much for the nature of our experience. To summarize, I talked about our use of PL/I: 1) in teaching beginning programming courses; 2) as a tool in other courses; 3) in a programming systems course where PL/I is the principal laboratory facility; and 4) in faculty and student research. I also mentioned smaller institutions who work completely via a Teletype.

Next, I should like to evaluate the language in terms of the needed improvements and the things which we have found to be advantageous. We will consider the needed improvements first and in order of importance.

The shortcoming we most severely feel is poor literature. The manuals are very bad. The main reference manual is complete and is rather accurate, but it is very had to find in it the answer to any particular question. The new version includes a certain amount of primer material at the beginning, which is a great improvement. The primer is a little better, but it is not a primer; it does not begin at the beginning and treat only the topics you need to know to get off the ground. The best primer is the publication IBM allowed to go obsolete and then out of print. It's called A Guide to PL/I for FORTRAN Users. Even if one had never touched FORTRAN—and I tried this on a class of students who had never seen FORTRAN—the Guide for FORTRAN Users was the clearest introduction to the language. There are some books beginning to appear; Mary Lasky here has one. These are beginning to help the situation. I would still say that from the point of view of use, this is far and away the most critical problem: the lack of good elementary material.

The compile speed of the OS/360 F-level compiler needs improvement. To some extent, this problem is peculiar to the university environment. If 300 students run 5 problems at 5 shots apiece, that is 7,500 compilations per semester. We want their compilations to go in a hurry, and we don't much care about execute time, because executions will be short and relatively infrequent. This is, of course, quite different from many production computing installations.

It is my opinion that any computing installation needs at least two versions of every important language: a fast-compile version and a fast-execute version. The two versions should be literally, strictly, exactly compatible. It is acceptable to have the fast-compile version be an interpreter, and I think as we move to more and more terminal orientation, this may be the solution. We

have available to us in FORTRAN the WATFOR system. WATFOR is an interpretive compiler written by the University of Waterloo, and it is very fast. If a job will run for more than 3 seconds, one doesn't want to use WATFOR because the object code is poor. On the other hand, student runs take about 3/10 second with WATFOR, when batched in batches of 10. PL/I takes us 8 seconds. With this discrepancy, Professor Ferrell at North Carolina State University is still teaching PL/I to his students in Chemical Engineering, but the people in the Computer Science Department are teaching WATFOR to their beginning engineers; they couldn't afford the difference. I am still teaching PL/I because when it's done in 8 seconds, we can afford it. But I very badly feel the need for a 1/2-second kind of compiler.

Operating System speed is another problem. The OS/360 scheduler takes longer than the whole WATFOR compiler, and the linkage editor takes longer than the whole PL/I compiler—this is ridiculous. However, we now have a loader built by American Data Processing which loads directly to core. For 95% of all jobs, this looks like the answer. I do not have any operating figures on the loader at this point.

Minimum object code size is a very serious problem on Model 30s. The object code takes almost 44K bytes of memory no matter how little the program, because of the subroutines. This is relatively painful on a 64K machine. On the 75 this is not painful, because big programs take little more space, and we operate in memory partition whose minimum size is 100K and whose maximum is 350K.

In the area of user conveniences, the most important is cleanup of the Job Control Language. Then, we need shorter diagnostics. The diagnostics are just plain wordy, and that's all right for a line printer, but when you're waiting at a Teletype, it just kills you to watch it go on and on and on. However, the diagnostics are in tabular form, and we have gone through on the Teletype system and just substituted terser remarks for some of these elaborate Anglicisms.

Another need is for less treacherous fixes. The compiler operates by fixing bad syntax so it can complete the compilation by hook or crook. For example, when it doesn't find a verb anywhere in the statement, it inserts an =, and I have never seen a case in which this helped. I've had it do this in the declarations, and thus it ripped away all the declarations. What happened then was glorious to behold: a cascade of diagnostics just bellowed forth, because all the arrays become functions, and all the subscripts then became meaningless or arguments, and it was just wonderful.

Interface with assembly program is awkward. We've been remedying this by writing some home-grown macros that our people could use. One such is just a standard PL/I enter; write it in the assembly program and it will put a prologue in front of it. We need something designed and supported to simplify interfacing in each direction.

The latest release looks as though it will overcome our principal problems with object code speed. The string manipulations were dead slow. The new release puts such in line, rather than going out to subroutines each time. This performs radical improvements in object code speed; it also makes the object look smaller.

I would say, in general, that of all the things that could be fixed, I would complain least about language problems, and put everything else ahead in the list. I have a list of language details, of course, minor points of one kind and another. The biggest one would be programmable defaults. The language originally defined had the ability for the user to specify what defaults he wanted, and I am eager to have that ability back; it is perfectly compatible with the language and could be done.

Now for the major pluses of the language, I think I've probably indicated them, but let me just summarize them. The first is the universality, the fact that we can, in fact, use it for teaching programmers from all parts of the spectrum, use it in all kinds of research, and build local expertise in one language instead of three or four. That has been the biggest plus in our shop, and of very great importance. The second is that it is easily subsettable. The default options mean that one can define subsets and the beginners do not have to know that other facilities exist. I like the controlled storage capability, the ability to manipulate what's happening in core. I think the macro feature is very important. I am excited by the ability to teach and to use compile-time processing as a routine part of a language. I think that the compile-time processing is going to be a very important trend in the development of programming languages; and I think that over the next five years we are going to see less emphasis on the development of procedure languages than on the development of subject matter languages. Then the question is, "What tools do I need to get good compilers for subject matter languages?" I think the answer is good compile-time facilities.

Then there are some advantages that will be mentioned many times today: the block structure taken from ALGOL, the whole control of scope and of storage allocation—and the control of scope and of storage allocation are properly

treated as two separate problems. The IF and DO facilities are far easier to use than in FORTRAN, for example. I asked all our instructors for comments, and every one of them mentioned the fact that the IF is a straightforward way of specifying conditions. The character and bit facilities have been mentioned. The I/O is rich, rich almost beyond bearing unless approached in subsets. The array and structure data types allow one to use the same kind of powerful tools that Iverson's notation allows for dealing with whole masses of data in one fell swoop.

Let me spend a minute or two on the F compiler per se, because it's very difficult to discuss this language apart from its implementations. The F compiler is rich in diagnostics. Our students find that they have no trouble taking the diagnostic output and determining their troubles from that. We teach students to take the source listing, the attribute table which tells what declarations the compiler used (including defaults), and compiler and execution error messages. We never take the object code listing, never take cross-reference tables, never take execute-time dumps, etc. We find that this is quite adequate.

The compiler's remarkably clean. The number of bugs per 10,000 instructions is the lowest of any part of the whole Operating System. The attribute table which I mentioned is very handy. Getting statement numbers in execute-time error messages is just elegant. This facility is the following: if, after I have compiled the code and put it away in the file, three months later I run it and I read an invalid card, or something, I get an error message. The error message includes the statement number from the source code as to when the error occurred, so I do not have to look at the object code listing. Debugging can, in fact, be done from the source listing. In my own experience, that has been a very powerful convenience.

Questions and Answers

Question: Your students are being exposed to other languages also, I imagine?

Answer: Mine aren't. They're being exposed to assembly language in the second course, and they're being exposed to ALGOL and FORTRAN by the time they get to about a fifth course, which is a programming languages course—but these are graduate students who are majoring in computer science. Most of the students are not being exposed to another language.

Q: Have you had any feedback from those who have been exposed to other languages, let's say before PL/I?

A: We get quite a few who have already come with FORTRAN experience, and we get some with COBOL experience. There are two reactions. One—and you get one or the other—is: "There's too much of PL/I," which means that we didn't do the subsetting job right; and the other reaction is the kind that we got from Professor Ferrell: "I'm glad I don't have to go back to the more rigid format."

Q: Are your comments on literature unique to PL/I, or do they hold for the 360 system in general?

A: The OS/360 system is highly variable in the quality of the literature. There are some things that are relatively well documented and some that are relatively poor. I would say the Reference Manual for PL/I is a very obscure document. It, however, is quite precise, and it is complete, but it is not easy to find what you want in it. In many of the control program manuals the material is imprecise or incomplete, and in many parts it is also obscure. Now, the Assembly Language Manual is a rather good manual. Some of the other language manuals are excellent manuals. You have to look at each one from the point of view of: Is it right? Does it tell the whole story? Can you read it? With respect to the PL/I Reference Manual, it seems to be right, it tells the whole story, and you can't read it.

Q: You said it's a good idea to declare all variables in a program and you also said that you'd like to have programmable default conditions, so I presume that you don't think it's necessary to declare all the possible attributes...

A: Absolutely not, absolutely not.

Q: ... so could you clarify which attributes you think ought to be declared, and which ought to be defaulted?

A: Well, in the first place, I neither know nor teach all the attributes for all the variables. I think it important to declare at least type, so: fixed, floating, character string. Obviously, for arrays you've got to declare the dimensions or the absence of dimensions. But,

from a programming practice point of view, I think it wise to declare all the variables. I teach students to declare them all at the beginning of the program so that the program becomes self-documenting. Notice in the illustrations I brought that I put comments in the declaration, variable by variable; the most critical thing to help a person read a program is to tell him what the variables stand for. So I really don't care how few attributes a student declares as long as he names the variables and puts comments by them.

I believe that the only solution to program documentation is to write self-documenting programs. Whenever you write the documentation independently, it's going to be late and out of date and not consistent with the program. So I have abandoned flow-charting completely and use a minimum of supplementary paper but insist that people put the documentation in the text of the program. That policy leads to this declaration rule.

Q: I noticed in your slides that the comments were side by side with the statements. This isn't the normal IBM compiler output, is it?

A: It comes out however you put it in. May I have the second slide again? I believe the most important comments in a program are the paragraphs you put first. The paragraph that comes first is the critical comment on the subroutine; it tells what it does, what its input and outputs are. The next most important comments, to my view, are the ones that come by the declarations, and it is very sensible for those to be side by side, variable and explanation, variable and explanation. Notice that I've factored the attributes; that is, I've said that J, P, and Q all have this single set of attributes. This is a facility the language allows—it saves labor—but I nevertheless put the variables on separate lines so that I can put separate comments by each one. I just fished this program out of the file and it has typo's—but this is the way this program is compiled and run in actual use; it has not been dressed up in any way. This is just the way it came out, and I submit that even if you don't know PL/I you can read and understand this program. It is a self-documenting program in that sense. Now, you will notice that I have given almost no comments here, and that's because this algorithm is straight out of the book. If you are interested in the details of the algorithm and can't follow it, there is a book reference for it. The first paragraph comment gets you off the ground; it is a standard binary search procedure.

Comment: (Professor Rosin) I want to save a lot of my comments for the panel, which I think is more appropriate. You said that you had very few objections to details of the language, with which I guess I would agree, but some of them appear to be central. There has been a great deal of discussion as to whether internal variables—variables internal to a nested procedure—should indeed be static or automatic by default. This appears to be one area in which a detailed language could become critical in terms of prevalent use. Is that possibly true in your case?

A: I think the right solution is programmable defaults.

C: I agree, but even in the light of programmable defaults, it seems to me that occasionally one would choose that the standard default—that is, one that need not be opted out—perhaps could be reconsidered.

A: No, in general I want automatic variables for nested procedures. Yes sir, I want that core to go away when I'm through using that subroutine so I can pull another subroutine in and not run out of space.

C: So it depends on how much core you have in the environment?

A: Automatic allocation is justified from the point of just good practice. Now, I would like the ability to write, at the beginning of the whole outermost procedure, "All allocation is static unless otherwise specified." The programmable default is the right way to solve that and a host of other problems.

C: It is my understanding that "implicit" is back in the language as a language point, although it is not scheduled for implementation. There were apparently a large number of problems of inheritance of attributes and priority of attributes; that is, if you said everything is character string, does that mean fixed string or does it mean varying string? Does it mean everything fixed, and does that apply to strings as well? There was a lot of consideration before "implicit" could be put back in.

A: I'm enough of a pragmatist and little enough of a scholar that if you tell me something's in the language and not in the implementation, I'm not really interested.

C: I agree with you, but I think it's important to know that it be considered.

PL/I IN A SCIENTIFIC ENVIRONMENT

Mary Douglas Lasky
Computing Science Group
Applied Physics Laboratory
The Johns Hopkins University
Silver Spring, Maryland

PL/I can be considered a logical development from previous major algebraic languages: FORTRAN, ALGOL, MAD and JOVIAL. PL/I has new features that make it unique and give reason to consider it as a major advance in programming languages, in a scientific environment, as well as for general use.

Evaluating PL/I's place in a scientific environment brings up several questions that need to be answered. Is PL/I adequate for the scientist's needs? Does it help scientists do things they could not do before? If so, what? What is the potential ability of PL/I in scientific computing? Is a general purpose language of more benefit to scientists than special purpose languages?

Programming in a scientific environment is roughly divided into two categories, that done by the professional who is programming large involved systems or general purpose library routines, and that done by the scientist who is programming his own research problems.

PL/I for the Professional Scientific Programmer

The majority of scientific programs are written in an algebraic or algorithmic or procedure-oriented language, that is, in a high level language that is relatively machine independent. However, if such an algebraic language proves inadequate, the professional programmer must resort to assembly language (i.e., symbolic machine language) to handle such problems as interrupts, storage management, character or bit string manipulation or critical speed of numerical calculations.

Furthermore, data or program errors will often raise a condition that can be detected by the program. The most common of these errors are overflow, underflow, zero divide, conversion, and such problems as a computed subscript exceeding its permitted array bounds or detection of an end of file. When such a condition is raised, it causes the program to be interrupted and control given to the operating system or monitor, and a standard system action taken. It is frequently useful to be able to specify an action to be taken in lieu of the standard system action. In PL/I this is done by means of an "ON" statement.

The "ON" statement has the general form of "ON condition on-unit". The "on-unit" is a PL/I statement or block of statements that specifies the action that the PL/I programmer wants taken when the "condition" occurs.

In general, other programming languages have no way of controlling the action taken when an interrupt occurs. If it is imperative that the system action be overridden, then assembly language routines have to be written.

The ability to handle interrupts becomes extremely important for large programs. Standard system action usually taken by the operating system is to write a message stating what happened and then terminate execution.

If a job runs for many hours, as for example, the satellite GEODESY program at the Applied Physics Laboratory, which runs for eight to nine hours at a stretch on the IBM 7094, then it is critical not to lose all the information that has been processed up to the point at which an interrupt condition occurs, and to try to recover to the extent that execution can continue or the job restarted. Otherwise a great deal of time and money is wasted. PL/I provides interrupt handling ability.

Storage management is another of the major problems arising when writing large programs. Several other algebraic languages do provide some method of storage management. For example, FORTRAN permits storage overlay of data and of program segments as does JOVIAL, but the arrangement of data must be carefully planned, and the process of overlaying segments is time consuming and impractical if many overlays are required. Storage allocation is static, and the amount of storage needed is fixed at compilation time.

On the other hand, dynamic storage allocation is the normal case in PL/I, and variables are automatically assigned storage each time the routine in which they belong is entered, and the storage released when the routine is left, thereby freeing space which is no longer required. This philosophy of storage management saves space and prevents some programs from ever getting so large that they would require overlaying of segments.

More important, this automatic allocation permits the programmer to assign and free storage for a given variable during execution. When planning large programs, this can be valuable. To accomplish this a variable is designated as "CONTROLLED". Then at any point in the logic of the program that variable may be "ALLOCATED", i.e., given space, and later "FREED". Later in the same program, that same variable name may be reassigned storage,

but this time the programmer has the ability to change the extents of the dimension or length of the string. As an example:

```
        ...
    DECLARE X(10) FLOAT CONTROLLED;
        .
L1: ALLOCATE X;
        .
L2: FREE X;
        .
L3: ALLOCATE X(50);
        ...
```

where "X" is designated as an array of 10 floating point numbers and assigned locations in storage when the statement labeled L1 is executed; the storage is freed when the statement labeled L2 is executed, and reassigned at the statement labeled L3 but this time as an array of 50 elements.

Automatic or dynamic handling of storage does have its drawbacks, because there is overhead in execution time associated with allocating and freeing storage every time a routine is entered. In order to save some of the time resulting from automatic or dynamic storage allocation, the programmer can specify that the variable is to be assigned storage for the duration of the program. This ability is also needed for variables that are used in several routines that are compiled separately but run together.

A third major problem area in scientific languages has been the manipulation of character and bit string data. Processing of information from satellites, missiles, and spacecraft, and from high speed experimental data acquisition systems, for instance, has become increasingly important in scientific computation. The data from these sources are not in numeric word form but, rather, a string of alphanumeric characters or a coded bit string. These strings need to be compared, scanned for patterns of specific bits, and character or bit strings inserted or extracted from the data. It is clear that character manipulation has become a necessity in the scientific environment.

PL/I provides for both character and bit string processing. These strings can be grouped in both arrays and structures; bit strings of length 1 can be used as Boolean variables. Powerful functions to manipulate string data are part of the language specification.

PL/I IN A SCIENTIFIC ENVIRONMENT

A fourth major problem area for the professional scientific programmer is speed of calculations. This seems to be a direct result of how the code is compiled. For example, if a large set of simultaneous equations must be solved repeatedly in the time that a satellite is passing overhead, or in the time a missile is in flight, there is no room for slack in any of the calculations. This has had to be done in assembly language before, and may have to be done in assembly language even with PL/I, as there are no features to take care of this problem. The faster machines may be the deciding factor in the quest for speed with high level languages. To maintain flexibility with an algebraic language, it is now both possible and economical to acquire a faster machine. This certainly was not true when FORTRAN was introduced.

Another problem of the professional programmer, particularly the one working on large programs, is to try to keep the program general so that changes will not be costly in reprogramming time.

At compile time PL/I provides facility, called the "compile time facility", to modify a program, expand sections of code, conditionally compile sections of code and include code from external sources.

This compile time facility provides the valuable ability to modify a program simply and quickly. For example, the precision attribute could be made a compile time variable. If the calcuations need to be more accurate, the precision can be extended by changing the value of the compile time variable, and thereby all the variables requiring the same precision are correspondingly changed. The conditional compile ability could be used in a large research program to selectively compile various sections corresponding to different experiments or to different theoretical models. Naturally, the programmer must be extremely careful when using this facility.

It should be noted here, with regard to desired future development of the language, that the compile time facility provided in PL/I, although valuable to the programmer, is not yet powerful enough to provide for introducing new graphics into the character set or changing the meaning of existing operators, such as "+", or "*", as a more general capability might. In order to make PL/I powerful enough so that specialized languages can grow from it, a more general ability is needed.

PL/I also provides for internal and external procedures. An external procedure is one that is compiled as a unit. Internal procedures are nested inside another procedure. Internal and external procedures coupled with interrupt handling and storage management make subprogram linkages complicated. The programmer

must be careful because organizing programs in PL/I in the same way they would be organized in other scientific languages may not be efficient. For instance, portions of programs put into subroutines in other languages might need to be kept in the main program to save the overhead. Probably each project should study this very closely and have a set of standards to be followed when forming large programs.

The interrupt handling, storage management, character and bit manipulation, as well as the compile time facility, give the professional programmer flexibility and power. Generally, no assembly language routines will be needed.

PL/I for the Research Scientist and for the Engineer

FORTRAN has been around for many years now; many scientists and engineers are familiar with the computer and know how to take advantage of it, to some extent, through a high level language. Since FORTRAN is relatively simple to learn, or at least to learn its basic principles, the scientist or engineer has usually had success with the computer.

There are some features in PL/I that will make it easier to use than FORTRAN and ALGOL, especially for the scientist doing his own programming. The main ones, I believe, are DATA and LIST Directed Input/Output. These are free forms of input and output and do not require format statements. In DATA Directed Input/Output, each variable name precedes its value.

For example, using DATA Directed Output, if "X" is to be written out and its value is 25, the output would appear as "X=25", thus, each value is easily identified. The LIST Directed I/O is equally easy and flexible. The COPY option on input statements gives a printout of the input values as they enter the computer.

On the other hand, there are several features that the scientist, who is not programming in PL/I frequently, will have to be aware of. The way PL/I passes arguments to subprocedures needs much study and can be confusing. In my opinion, PL/I has been made needlessly complicated in this aspect. Constants used as arguments can be pitfalls particularly. I believe it is a good idea to always have a DECLARE statement for the subprocedure being called and explicitly declare the attributes for each argument. For example, if procedure SUB is to be called and it has arguments A and B, which are floating point and fixed point respectively, then the DECLARE statement should read:

```
DECLARE SUB ENTRY (FLOAT, FIXED);
```

The scientist must also be aware that the language is not yet smooth in all respects. There are places where in one operation an expression may be used and in the next operation a similar expression is not allowed. For example, the precision for the ADD function must be a constant while expressions are allowed for the variables to be added. Here again, I see no reason for not permitting an expression for the precision.

However, the research scientist not only benefits from the flexibility of PL/I in describing complicated processes, but also, where many specialists are working on the specifications of a complicated system (such as the satellite system at the Applied Physics Laboratory, mentioned before, which required assembly language programming) it is now possible for each scientist to develop his own component in PL/I and have the professional programmer assemble the system directly, and in a form intelligible to the scientist.

Do the New Features in PL/I Truly Help in Solving Numerical Problems?

Problems in the actual numerical calculations, as contrasted with the mathematical formulation, are very important in scientific computing. The question is asked, "How accurate are the numerical answers to the problem?" Numerical difficulties in the results may be due to instability of the technique, or they may be a matter of precision. (They can be a result of both.)

Precision in PL/I can vary from "1" to "n" decimal or binary digits. The "n" will, of practical necessity, be set at an implementation maximum. An example of the precision attribute is:

```
DECLARE X FIXED DECIMAL (8);
```

X is a fixed-point decimal number with 8 significant digits.

FORTRAN, typically, is implicitly word machine oriented and only single and double precision, rather than a range of precision, are provided. The wide range of precision in PL/I has great potential for studying different ranges of accuracy in a problem.

A great deal of effort has gone into the error analysis of numerical computations, and one way to prevent errors from accumulating in critical operations is to use extended precision and then truncate or round for the final answer. This means

that twice as much storage is taken for each variable. One of the most critical error problems arises in inner products. If these alone can be taken in extended precision, such numerical calculations are greatly improved. The error produced from doing the summing of inner products in single precision is approximately $(1/2)b^{1-t}$, where b = base and t = number of base b digits, whereas the double precision error becomes $(1/2)b^{1-2t}$, which is virtually equal to 1, for there is only truncation or rounding of one unit in the least significant digit of the single precision number (Reference 1).

PL/I provides two built-in functions, ADD and MULTIPLY, which allow for addition and multiplication in any desired precision. To multiply two vectors of n elements each and add them to an extended precision number, H, using the ADD and MULTIPLY functions for n numbers, we might state:

```
DECLARE (X, Y) (N) FLOAT,
          H FLOAT(k) INITIAL (0);
DO I = 1 to N;
   H = ADD(H, MULTIPLY (X(I), Y(I), k), k);
END;
```

The "k" is a constant expressing the precision of the multiplication and addition function. For large or for very sensitive calculations this can be of advantage.

Some, but not all, compilers for other algebraic languages automatically use extended precision if the variable on the left-hand side is in extended precision. If a compiler does not, then either extra variables of the extended precision have to be introduced or assembly language routines need to be written to obtain similar results. The PL/I functions insure how the precision is to be carried out.

I took two numerical problems to do a limited study in order to get some feeling of the benefit gained from the extended precision and from the ADD and MULTIPLY functions. The first problem is the solution of systems of simultaneous linear equations. Two different, commonly used techniques were chosen for comparison, the traditional method of Gaussian elimination with pivoting, and the Crout decomposition with pivoting, the latter having potentially less error when inner products can be accumulated in extended precision (Reference 1).

I also chose a modified Newton method for finding zeros of polynomials because it involves sums of inner products in a complex mode (Reference 2).

Since this was a limited experiment no absolute judgements should be based on it, but the following conclusions seem correct.

1) It is of definite advantage to be able to use a compile time variable to specify the precision when doing experiments of this type. The program can be easily modified to obtain more accurate or less accurate results.

2) The ADD and MULTIPLY functions did help considerably for ill conditioned forms such as the Hilbert matrix.

3) Gaussian elimination in the single precision implemented on the IBM 360 lost a great deal of accuracy, but was greatly improved by using extended precision.

When programming basic library routines such as these, it is valuable to be able to keep them as general as possible. The bounds of an array in a called procedure can be specified by asterisks, *, which means that the value of the dimensions will be taken from the calling program. For example:

```
MAIN:       PROCEDURE OPTIONS (MAIN);

            DECLARE (X(5), Y(-2:7), Z(100)) FLOAT;

            CALL SUB (X);

            CALL SUB (Y);

            CALL SUB (Z);
            ...
END MAIN;

SUB:  PROCEDURE (A);

      DECLARE A(*) FLOAT;
      ...
END SUB;
```

The variable A in the procedure SUB takes on the bounds of the argument from the calling procedure. When SUB is called with X as the argument the bounds of A are 1 to 5, with Y the bounds are -2 to 7, and with Z the bounds are 1 to 100.

As has been shown in the example, the upper and lower bounds of the dimension can be any value as long as the upper bound is greater than the lower bound. The

"*" for the bounds in the calling program is unique to PL/I. Although other languages have ways of accomplishing the same results, they are not clear and disguise what is truly happening.

FORTRAN Libraries

Nearly every scientific installation in the country has a FORTRAN compiler, and nearly every general purpose computer has a FORTRAN compiler written for it. Large libraries of routines are written in FORTRAN, and consequently, a great deal of interchange of programs is done in a scientific environment.

Unfortunately, since there are many dialects of FORTRAN, this interchange is not always as smooth as it could be. Different compilers handle operations differently, and what runs under one system may not run under another. There is a standard USASI FORTRAN, but this is not always adhered to. Also, programs may include assembly language routines which may need to be rewritten.

The interchange of programs is not as frequent in the commercial world because the programs are often tailored to an individual installation. So far, PL/I is being used more in the commercial realm than in the scientific. I believe that these large libraries already built up in FORTRAN have been largely responsible for people's hesitation to move to PL/I. There is a large reprogramming job ahead and communication between languages is not trivial at present.

At the Applied Physics Laboratory, our problem is somewhat different, because the majority of our programs are either in assembly language or in FORTRAN II. Consequently, we must necessarily do language conversion to go to "third generation hardware". All of the programs for the Satellite Division, which uses the majority of the time on our computers, are written in assembly language because FORTRAN was not adequate for their needs, but PL/I satisfies their needs. Therefore, we at the Applied Physics Laboratory are converting to PL/I.

We have written a FORTRAN II-to-PL/I translator, written in PL/I, which took eight people about one man-year to produce. We did it to gain experience in PL/I, and to learn something about writing large systems in that language. The translator is being released to the SHARE Distribution Agency.

As part of our study of realistic scientific applications, we selected a program used by the Aeronautics Division at APL, which was originally written in

FORTRAN II at the Naval Ordnance Test Station with the title "Computation of Equilibrium Composition Thermodynamic Properties and Performance Characteristics of Propellant Systems" (Reference 3). The program was modified at APL and is used to determine the composition and thermodynamic properties of combustion products for rocket and ramjet performance estimates. I believe that a certain variation of the same type of study is done by the Air Force. The mathematical method used to solve the set of simultaneous equations is Villar's method, which is iterative and does not require matrix inversion. The translation of the program was started at the end of October 1967, and we hope to report on the results in the near future.

Other PL/I Features

So far, I have concentrated on advances PL/I has made that are especially useful to the professional programmer and to the research scientist. There are other features of PL/I that have advantages in scientific applications. Some of them are structures, list processing, asynchronous operations and increased input/output facility.

Structures provide a powerful technique for grouping nonhomogeneous data, and I believe that as the scientist becomes more familiar with them he will find structures invaluable. For example, in satellite tracking, what is known as the satellite epoch is best defined as a structure containing the year and day of an epoch as one type of data, and seconds as another type. All three elements may be treated as a unit "EPOCH" as follows:

```
DECLARE 1 EPOCH,
          (2 YEAR,
           2 DAY) FIXED BINARY,
           2 SECONDS FLOAT;
```

List processing is used in scientific work for such things as report generation, and it is valuable that the capability is provided in an algebraic language.

Asynchronous operations are also provided in order to take advantage of these advances in technology. A task can be attached during execution, then the tasks can be synchronized or tested for completion. This overlap of execution time has great potential.

More and more data are being generated by scientists, and it becomes increasingly important to be able to store and retrieve the information rapidly in machine-readable form. After such data are stored, not all of them are needed at any one time again. Therefore, the data need to be searched and sorted, etc. The need to handle this efficiently in an algebraic or procedure-oriented language is important.

PL/I provides for large volume input and output by means of "LOCATE mode Input/Output". Data are located in the operating system's buffers and are not transferred from the system buffer to the program storage area unless they are needed for actual calculations. This is efficient because there is not a core-to-core move for every record (Reference 4).

Conclusion

There is a growing feeling in the computer community that special purpose languages can best solve the needs of the scientist. On the other hand, there is just as much feeling that the trend is toward general purpose ones. PL/I is a general purpose language, and I believe it can be of great benefit in scientific computing. There has not been much interchange between the commercial and scientific worlds in the past. As the scientist's need for large volume data increases, he can learn much from the commercial people who have been dealing with these problems for a long time. In return, the scientist can teach the commercial people about the handling of numerical calculations. With a common language this will happen more readily.

As far as the scientific environment is concerned, the advances in PL/I will aid the professional programmer to solve his problems more easily, with the ability to handle interrupts, manage storage, manipulate bits and character strings and handle input/output more effectively. The research scientist will have a simple input/output capability and a way to express his problem so that it reflects his research model. Since both types of programmers will be using the same programming language they can more easily communicate.

PL/I has made a significant advance in computer languages and will be an advantage to the scientific community both for large scale systems and for research problems.

References

1) Forsythe, George and Moler, Cleve, Computer Solution of Linear Algebraic Systems. Englewood Cliffs, N.J.: Prentice Hall, Inc., pp. 148, 27-31, 47-48.

2) Ehrlich, L. W., "A Modified Newton Method for Polynomials," Communications of the Association for Computing Machinery, 10 February 1967), pp. 107-108.

3) Cruise, D. R., "Notes on the Rapid Computation of Chemical Equilibria," The Journal of Physical Chemistry, 68 (December 1964), pp. 3797-3802.

4) Bates, Frank and Douglas, Mary L., Programming Language/One. Englewood Cliffs, N.J.: Prentice Hall, Inc., pp. 302-306.

Questions and Answers

Question: If you were were writing a PL/I program and you wished to get into a subroutine that is already written in assembly language, is there any convenient way that you can write this into your PL/I program, go into it, operate in the assembly mode, and then at the proper time return to the PL/I program?

Answer: Yes. We have done some work along this line. We are groping along with many other people in this whole area of inter-language communication. If you know how PL/I is set up, then you can fairly easily write your assembly language routine. You must know how your arrays are stored and whether your calling sequence refers to the data item itself or to what is commonly called a "dope vector," which is just a way of describing the data. We have not had many problems in communication if the main routine is in PL/I—but if the assembly language routine is the main routine and calls a PL/I subroutine, more problems arise because of the way PL/I handles interrupts. Let me recommend that if you can have a PL/I main procedure, then your job of communication between languages is a lot easier.

Comment: (Professor Rosin) Let me offer a footnote here. There is a programmer at our installation who was assigned the task of writing an assembly language program to support the 2260 device, which is a typewriter character display tube. The installation has no standard language at present. What he did, relatively easily—I think it was probably easier to do this than any other of the alternatives which might be available—was to write a common routine with alternative entry points for use in PL/I, FORTRAN, and COBOL. The need to provide a common routine meant that he had to standardize a lot of things which otherwise would have been done in particular ways. The result is a very effective and efficient program.

Q: Does PL/I handle the external interrupt—let's say from I/O—in the same manner as the internal?

A: These I/O interrupts interface mainly into the operating system. My experience has been with writing a 2260 routine which is called by a PL/I program—I wasn't so clever as to interface it with all the languages. My technique was to have the assembly language routine wait to get the trap from the 2260.

C: (Professor Rosin) A fellow at Bell Labs has written a very effective routine that generates a signal which is indeed an asynchronous interrupt. There is a problem of PL/I handling true asynchronous interrupts which isn't clearly resolvable right now, but it is possible. The Bell Labs program is written as a library routine, at this point in assembly language. I would hope eventually to find a library of routines that will provide facilities which interface rather directly with PL/I—so that one could say, for example, "On 2260 interrupt do thus and so," or "On 2260 interrupt give me the name of the 2260 which generated this interrupt."

IBM is now considering—with SHARE's instigation, guidance, and sometimes thorns—the integration into PL/I of facilities which will allow these routines to be written more directly into PL/I. As far as I've been able to tell from my limited survey, people have been successful in providing this kind of support—and in a way which is more and more compatible with PL/I—taking advantage of the PL/I interrupt facilities and ON-code, ON-unit, and other pseudovariables that can be used.

Q: In your paper you indicated that one of your large programs at APL runs 8 to 9 hours on the IBM 7094. I'm sure you have considerable concern, in rewriting this in PL/I, about how fast it will run on whatever 360 you are using—Model 65?

A: We hope to get a Model 91 in July, and have test time starting in January. We don't have any real results of how fast PL/I routines are actually going to run.

Q: Have you made any estimates, forecast memory requirements and run time relative to the 7094, given some scale of performance between the two computers?

A: Only a small portion of the geodesy program you asked about has been written in PL/I as a research project. We are keeping one 7094 so we can run these problems in their present form for some time. Also, there's been a great deal of new physics developed, so that when the programs are completely rewritten it won't be a matter of straight translation.

Q: Do you have any statistics on translating some of the problems from the 7094?

A: One program that I mentioned—done by the Aeronautics Division for ramjet and rocket analysis—has a problem in the amount of core generated; it used between 16 and 18K on the7094, not counting the operating system. After we translated it to PL/I it takes 28K words on the 360. This is a line-for-line translation from FORTRAN II to PL/I; that is, no improvements were made to make it good PL/I code. For example, the program requires large arrays so that it will satisfy any case desired. With dynamic allocation in PL/I combined with the asterisk notation you saw in one of my examples, the storage requirements can be cut down in most cases.

C: (Professor Rosin) I'd like to add something that may go further than what Dr. Brooks said about how clean the PL/I compiler was for the first cut at a very large scale language processor. I think it's interesting that we always compare the current PL/I processor with processors for languages that have been around for a long time. I suspect the second try at building a large-scale PL/I

processor will result in a smaller requirement for object-time space and in better object-time efficiency—perhaps by orders of magnitude. It's a little unrealistic to compare the first cut at something this large with processors for a language like FORTRAN, which has been around for 10 years.

THE UTILITY OF PL/I FOR COMMAND AND CONTROL PROGRAMMING

Christopher J. Shaw
Project Leader
Information Processing Technical Information Center
System Development Corporation
Santa Monica, California

Abstract

This paper: attempts to define and characterize the command and control application area; discusses the requirements this application area imposes on programming languages and processors, and how well PL/I meets these requirements; comments on PL/I's prospects for widespread use in the command and control area; recommends that the Air Force command and control programming language standard not be changed from JOVIAL to PL/I until significant savings in time or money can be demonstrated.

Any discussion of the utility of PL/I for command and control programming—its advantages and weaknesses—must be fairly academic, since no command and control programs have as yet been written in the language. Nevertheless, the command and control application area is now more than ten years old; the kinds of programming tools required for it are well known, as are the demands it makes on a programming language. And PL/I is available for study and use, so it is possible to form some relevant opinions on the question of whether PL/I is really suitable for command and control programming.

In the most general terms, it is clear that the answer to this question is "yes." PL/I is a very broadly applicable language, and command and control is a very wide application area, so there is bound to be some overlap. But this observation is essentially trivial. What is needed are some practical guidelines for looking at a particular command and control system to determine its programming language requirements, and for looking at the various alternatives—including dialects of PL/I—to make the best choice by matching the requirements against the alternatives.

THE UTILITY OF PL/I FOR COMMAND AND CONTROL PROGRAMMING

There are presently only two practical alternatives to PL/I for command and control programming: assembly language and JOVIAL. I have no detailed guidelines to offer here on how to choose among these alternatives; that is a topic for another paper. This paper deals with a few, more basic topics. First, what is command and control? Everyone has his own ideas, of course, and there is probably even a standard Air Force definition. Nevertheless I should explain how I look at it, so I can talk about the kind of programming language you need to write a command and control system, and how well PL/I meets these needs. Finally, I have some comments on the question of PL/I's prospects in command and control.

What is Command and Control?

The Definition

A military command and control system can be defined as an information-processing system that is intended to provide military personnel with direct computational support in conducting an actual or anticipated battle. This definition includes all of the command and control systems I know of, but it excludes many military systems that are not used in fighting battles, such as payroll accounting systems, and it also excludes systems that do not have a man/machine interface, such as weapons guidance systems. These exclusions, however, do little to delimit the kind of language you need for command and control programming.

The Spectrum

Looking closely at the command and control application area, you find it covers a more or less continuous spectrum of applications. This is shown in Figure 1.

Unfortunately, there are two kinds of people in the world: those who explain everything in terms of dichotomies, and those who do not. Thus the command and control application spectrum is often said to consist of only two major categories. These have had many names applied to them over the years—names like "tactical" versus "strategic" systems, or "real-time" versus "nonreal-time" systems. For simplicity, let me call them: "surveillance and control-oriented" versus "command-oriented" systems.

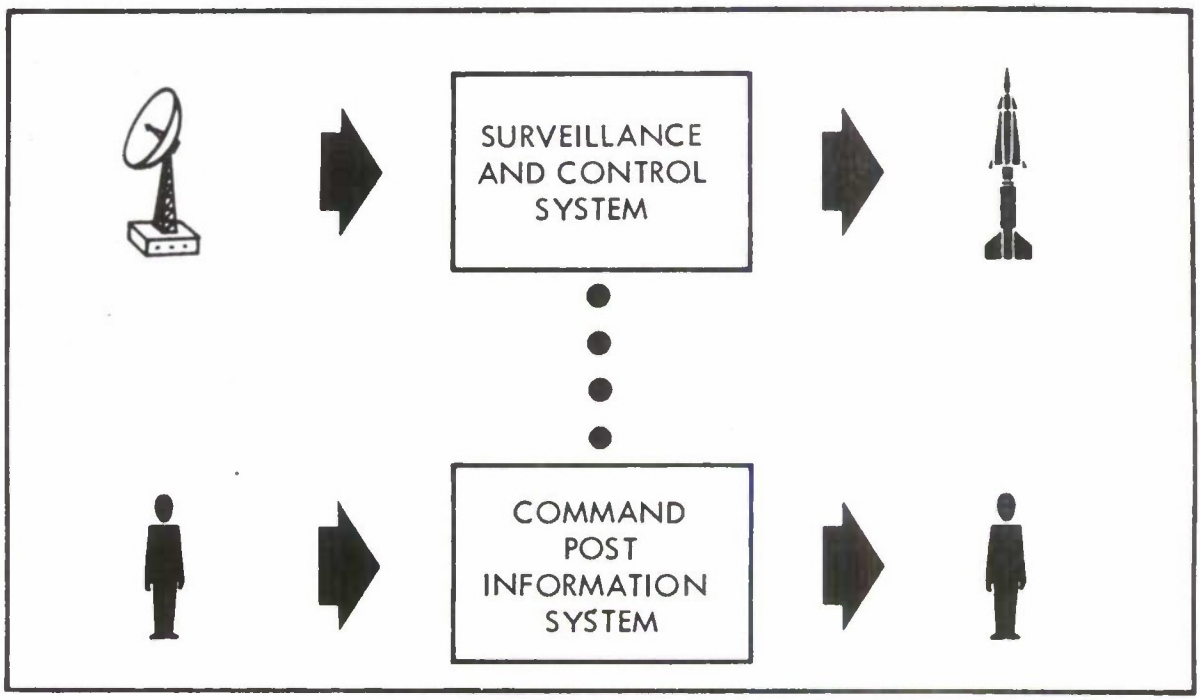


Figure 1. The Command and Control Spectrum

This kind of distinction is commonly made, but it is the basis for the distinction that is important, and hard to pin down. I think it is the degree to which the system is automatically linked to the outside world that it must sense and affect. Now this is something you can measure. You can measure the ratio between the system inputs that come from people, and the inputs that come from some kind of automatic sensor, like a radar. And you can measure the ratio between the system outputs that go to people, and those that go to some kind of automatic control system. So you actually do have a spectrum.

At one end of this spectrum, there are surveillance and control systems, such as the Ballistic Missile Early Warning System (BMEWS) and the Sentinel anti-ballistic-missile system, which sense and affect their environments almost completely automatically. At the other end, there are command post information systems, like the U.S. Strike Command's STRICOM system and the Air Force Headquarters' 473L system, which sense and affect their environments manually. In a surveillance and control system, response time and reliability are the key things, and human interaction tends to be minimized and fairly rigid. For many reasons, such systems usually cannot afford too much computational inefficiency. In a command post system, there are still response

time problems, but they are not nearly as important. (A missed deadline does not automatically mean a system failure, it just means an irritated commander.) Human interaction is the main thing in this kind of system, so a lot of computational flexibility is needed.

Then in the middle of the spectrum, there are command and control systems where the surveillance, control, and command functions are combined. Perhaps the SAGE air defense system is the best-known example here.

The Software

Let us consider briefly the kinds of computer software necessary for a command and control system—or any system, for that matter. This is another area that has fallen into the hands of the dichotomizers, who currently use the labels "functional" and "nonfunctional" software. This is unfortunate, because it will be hard to get managers to buy "nonfunctional" software, even though it forms a vital part of a system.

For most purposes, software is better categorized as utility, application, or support software, as shown in Figure 2.

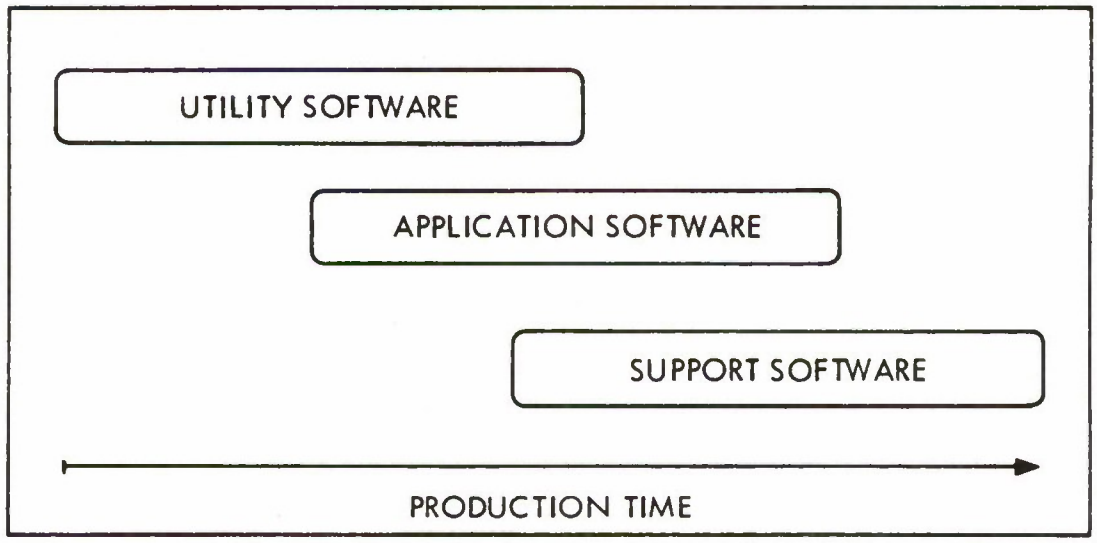


Figure 2. Command and Control Software

THE UTILITY OF PL/I FOR COMMAND AND CONTROL PROGRAMMING

A utility program is a programming tool used for producing, testing, or installing other programs. Examples are compilers and assembly programs, debuggers, test data generation programs, and the like. Operational or application programs are those that help the military users do their command and control job. Nobody doubts the need for this kind of software.

Finally, there is the vital yet sometimes overlooked category of support software. A support program is one that is used in training and exercising the system, particularly its human operators, or in analyzing and evaluating the system, in both live and exercise situations. A support program necessarily deals with the whole system, including the people. A program used to exercise and evaluate just the hardware or just the software would be a diagnostic or a utility program, rather than a support program. Some examples are environment simulation programs for producing exercise inputs, and operational recording and analysis programs for auditing or determining how the system is doing.

These three categories of software impose their own requirements on a programming language, in addition to the requirements of any particular type of command and control system. Let us look at some of these requirements, at least the unusual ones—those that are not satisfied by languages like ALGOL or FORTRAN—and try to see why they are almost always necessary.

Command and Control Language Needs

Aside from the ordinary features every programming language needs, the essential requirements of a command and control programming language are few. They include the capability to specify text processing, bit processing, fixed-point arithmetic, arbitrary data origins, varying table entries, and preset data. Interestingly enough, these requirements all have to do with data types and structures.

Text Processing

A command and control programming language must be able to handle text processing, by which I mean arbitrary manipulations on arbitrary character strings. In most command post information systems, at least part of the data base is textual; text is what the command staff puts into the system and hopefully what they get back out of it. Even in surveillance and command systems, there is usually some textual communication with the operators, even where most of it

is by function-push-button or light-pen actions and graphic displays. In addition, communication between the utility and support systems and their users is largely textual.

These systems can be programmed in a language with a very rudimentary text-processing capability. Languages as elaborate as SNOBOL, for example, are not really necessary, although extra features are always helpful if the user can afford what they cost. What is necessary is the ability to declare textual variables of arbitrary length—up to some reasonable maximum. The ability to declare textual variables up to the length of at least a print line is useful, because this simplifies much of the programming. But textual variables limited to the length of a computer word, say 6 or 8 characters, are adequate. Other requirements are the abilities to assign new values to textual variables, denote constant textual values, and compare two textual values for equality and alphabetical order. A concatenation operator is useful, though not vital. However, it is essential to be able to designate, as a textual variable, any dynamically specified substring of another textual variable.

The following examples show how PL/I expresses some of the text-processing functions I have mentioned.

```
declare TEXT character (500) varying;

substr (TEXT, I, 3) = '('||A||')';

if 'M' <= substr (TEXT, I, 1) then . . .
```

Fixed- or varying-length textual variables can be declared in PL/I. In the examples, a variable named TEXT is declared to have a maximum length of 500 characters. (In the IBM F-level compiler, variables up to 32,000 characters in length can be declared.) Textual substrings can be dynamically specified in PL/I. In the example (second line), the three characters beginning at the Ith character are being assigned the value: a left parenthesis, concatenated with the (presumably single) character designated by the textual variable A, concatenated with a right parenthesis. The third example shows the Ith character being compared to see if the letter "M" is less than or equal to it.

Of course, PL/I has other text-processing features. In particular, it has a very useful string-searching procedure named index, and another one named length that indicates the current size of a string. But these are extras, not really essential for command and control programming.

THE UTILITY OF PL/I FOR COMMAND AND CONTROL PROGRAMMING

Bit Processing

Just as text-processing features are needed to write programs that communicate with people, so bit-processing features are needed to write programs that communicate with the remote sensors, control devices and other systems that a command and control system must interface with. Since these devices tend to communicate in strange codes and formats (that language designers evidently have never heard of), it is necessary to manipulate individual bits. However, this is not a universal requirement, because in some command post systems the interfaces may all be textual. But it is common enough, and if new equipment requiring new formats is added to an existing system, it is useful to be able to program for it.

In essence, the bit-processing capabilities required are identical to the text-processing ones, except they are applied to bit strings rather than character strings. In addition, it is desirable to do logical operations on bit strings—that is, treat them as Boolean vectors. But again, this capability is not vital. Here are some examples of bit processing, dealing with a 32-bit string named X.

```
declare X bit (32);

substr (X, 1, J) = ¬(A & B) | C;

if substr (X, I, 1) = '1'b then ...
```

In PL/I, bit strings are handled like character strings, with the same operations. In addition, all the logical operations on bit strings can be used, such as "not" (¬), "and" (&), and "or" (|). In the second example, the first J bits of X are set to the resultsof "and"-ing the bits in A and B together, taking the complement of that, and "or"-ing it with the bits in C. The third example tests the Ith bit of X, and shows how awkward the PL/I notation can be when compared with the nearest equivalent ALGOL expression shown below.

```
if X[I] then ...
```

Fixed-Point Arithmetic

A capability for fixed-point arithmetic is mandatory for command and control, mainly because of the economy in execution time and storage space it provides

(compared to floating point). In most computers, fixed-point arithmetic is much faster than floating point, and this can make a vital difference in a real-time system. In any kind of system, it takes a lot less space to store, for example, 5,000 fixed-point numbers each 18 bits long than it does to store 5,000 floating-point numbers that are each 36 bits long. Also, less time is spent managing this data, moving it in and out of core. While this difference may be unimportant in a prototype command post system, for example, where floating-point numbers are acceptable for sample files, in any real system, floating-point numbers may be a luxury the user cannot afford.

To do fixed-point arithmetic, the only essential requirement is the ability to declare fixed-point variables of arbitrary size and scaling. That includes the ability to declare part-word variables, but not necessarily multiword variables, and the ability to declare the radix point to be anywhere inside—and even outside—the field boundaries of the variable. Also, of course, it is necessary to be able to denote constant fixed-point values. Naturally, when arithmetic on fixed-point values is specified, the compiler should automatically take care of the scaling.

PL/I has almost all the facilities for doing fixed-point binary or decimal arithmetic, as shown in the following examples.

```
declare (X binary, Y decimal) real fixed (8, 5);

X = 1.1b; Y = 10+(3/2);

A: if X = Y then go to A;
```

The first line of code declares X and Y as 8-digit, fixed-point variables, each with 5 fraction digits—binary in the one case, decimal in the other. The "5" in the declaration could have been just about any positive or negative integer. (In the IBM F-level compiler, binary variables can be declared up to 31 binary or 16 decimal digits long.) Still, there are some minor but very real annoyances. For instance, all fixed-point variables in PL/I carry sign bits, and a positive, unsigned variable cannot be declared without a sign bit (although an unsigned binary integer can be declared as a bit string). Even though it looks as if part-word variables could be declared in PL/I (such as X, which is nominally just 8 bits plus a sign bit long), the F-level compiler does not really allow this, because it allocates storage for fixed-point variables in full-word units.

Another problem concerns the fixed-point scaling rules in PL/I, which sometimes produce curious results. The second line of code in the examples above sets X to the binary constant one and a half (1.1b), and Y to the formula 10+(3/2), which—incidentally—could just as well involve variables with the same precisions and values. Then, barring any conversion errors I have not bothered to account for, you will find that X does indeed equal Y, because the division (3/2) gives you the maximum number of digits, but only one integer digit. To add the ten, you must truncate something, and the PL/I scaling rules demand truncation of the most significant integer digit, rather than the least significant fraction digit.

Arbitrary Data Origins

Another command and control language requirement is the ability to declare origins for data elements. That is, the programmer must be able to say that a certain item starts in bit so-and-so of word such-and-such, both in absolute terms, and relative to some other location, and with provision for overlaps and gaps. The command and control programmer needs this so he can deal with the irregular data formats in the unusual places he is sure to encounter, particularly in a real-time system. At times he must be able to get at data that is wired into certain bits in core memory.

One way the programmer might use PL/I to get at any given string of bits is to declare an array that takes up all of core memory, which a "smart" compiler could assume was supposed to overlay everything else, including the program. Here is an example of this:

```
declare WORD (32768) bit (32), FIELD bit (20)

  defined WORD(639) position (8);
```

Assume you have a computer with a core memory of 32,768 words, each 32 bits long, and you want to get at bits 8 to 27 of word 639. The first part of the example above declares the WORD array; the second declares a 20-bit FIELD, defined to start right where you want it.

One problem with using defined overlays is that the only thing that can be defined on top of a bit string in PL/I is another bit string. The same holds true for the other data types. Now, let us say you want to describe the format of a word, as shown in Figure 3, where the first bit is an indicator (named I)

that tells you whether the rest of the word contains a regular 8-bit character (named A), or a character (named B) from some special, 5-bit alphabet such as the Baudot teletypewriter code.

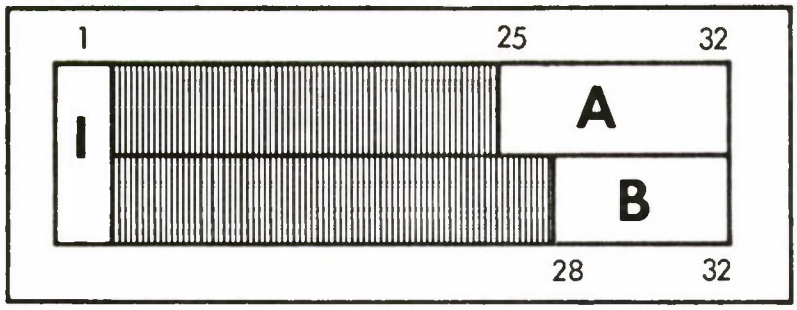


Figure 3. A Simple, One-Word Data Format

The format in Figure 3 can be declared by the following "cryptogram."

```
declare
1 NULL1 packed,
   2 I bit (1),
   2 NULL2 cell,
      3 NULL3,
         4 NULL4 bit (23),
         4 A character (1),
      3 NULL5,
         4 NULL6 bit (26),
         4 B bit (5);
```

Notice that this simple data format is declared in PL/I as a four-level data structure, with half a dozen null names (which have to be there, and have to be unique—at least those at the same level have to be unique) even though we will never use them in the program. At the first level, we declare the whole word as a packed data structure (NULL1), which means there is to be no unused storage between adjacent data items. (Unfortunately, this only applies to string items in PL/I.) At the second level, we declare a 1-bit indicator (I), and we also declare a cell (NULL2), which is an area of storage with several alternative formats. Ours we declare as a pair of third-level structures (NULL3 and NULL5). One (NULL3) consists of the character item (A), preceded by a 23-bit filler item (NULL4), which we need so that A will start at the right place. The other (NULL5) consists of a filler item (NULL6) followed by the 5-bit character item (B), which we have to declare as a bit string, though we will probably process it as an integer.*

The previous example shows it is possible to declare rigid data formats in PL/I, even though the language really is not designed to make this easy. It is also possible, though awkward, to declare tables with varying entries in PL/I.

Varying Table Entries

A typical data structure in command and control applications is the table with variable-size entries. An example of this is a table of flight plan data, where each flight plan has a variable number of checkpoints. Figure 4 shows a very simplified example of an entry in such a table. It assumes that the common data items for each flight plan fit into one 32-bit word. The n checkpoints ($P_{379,1}$ to $P_{379,n}$) are each ten bits long, and are stored three per word. Presumably, they are indexes to some other table (of location identifiers). The flight plan table itself would be indexed by word, so that $P_{379,3}$ would refer to the third checkpoint of the flight plan entered at word 379 of the table.

There are ways to avoid the varying table entry, but it is so common and useful a structure that any command and control programming language should be able to handle it. PL/I can handle it, in fact, though not, as you might expect, with the varying attribute, which results in the allocation of a fixed,

*It would often be very convenient if PL/I allowed programmers to define arbitrary alphabets (see Reference 4), but it does not.

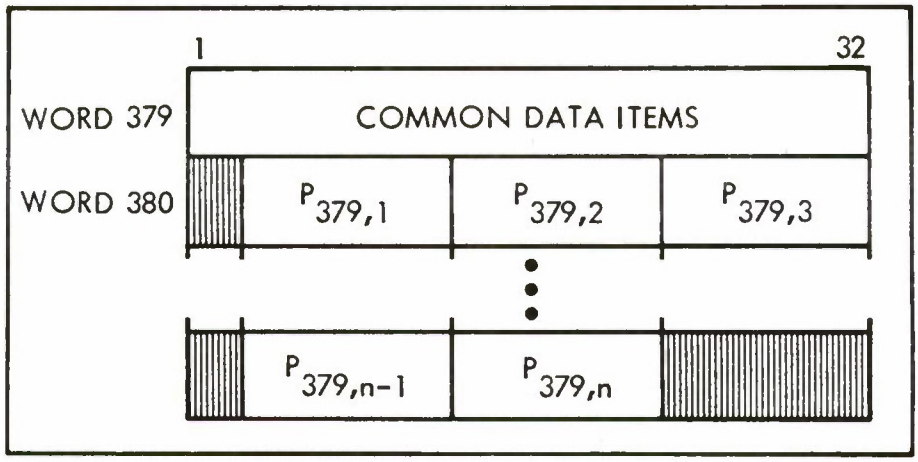


Figure 4, A Variable-Size Table Entry

maximum amount of storage. One way this can be done is shown in the next "cryptogram."

```
declare
1 FPT (2500) packed cell,
   2 NULL7,
      3 ... ,
      : /*common data items*/
   2 NULL8
      3 NULL9 bit (2),
      3 Q (3) bit (10),
P (2500, 3) bit (10) defined Q(1+1sub+ floor(2sub/3),
   1+mod(2sub , 3));
```

I have named the flight plan table FPT, and allocated 2,500 words to it. It is a packed cell with two alternative, second-level formats. (Unfortunately, this is a coding mistake. For some reason, you cannot declare a packed cell in PL/I; the two attributes are incompatible. But you can easily get around this restriction by inserting another structure, with the cell attribute, between the first-level structure and the two second-level, alternative structures.) The first one (NULL7) contains the word full of common data items, which I have not bothered to spell out. The other (NULL8) contains a 2-bit filler (NULL9) and three checkpoints, which I have had to give a dummy name (Q), since I can not index them the way I want to index the P's. Thus, I must declare P as a separate array, with dummy (but hopefully adequate) dimensions, defined onto Q. The complicated subscript expression after Q gives the functions for mapping the subscripts of P—which are referred to as 1sub and 2sub—onto the subscripts of Q.

Preset Data

The last requirement on my list is the need to initialize data within the program at compile time. This is a common requirement, and not just for command and control programming. There are many uses for such things as tables of constants, and while there are other ways of making them available to a program besides having the compiler generate them, these are usually rather cumbersome in actual practice.

PL/I has a reasonably good mechanism for initializing data values. The following example gives the declaration for the rather picturesque character array shown in Figure 5.

```
declare TAB (8, 3) character (1)

   initial ('A', 'B', 'C',

   (18) (1) '*',

   'Y', 'Z', ' ');
```

A repetition factor of 18 (on the third line above) is used to get the 18 asterisks. But to avoid producing a single, 18-character string, another repetition factor of one must be inserted.

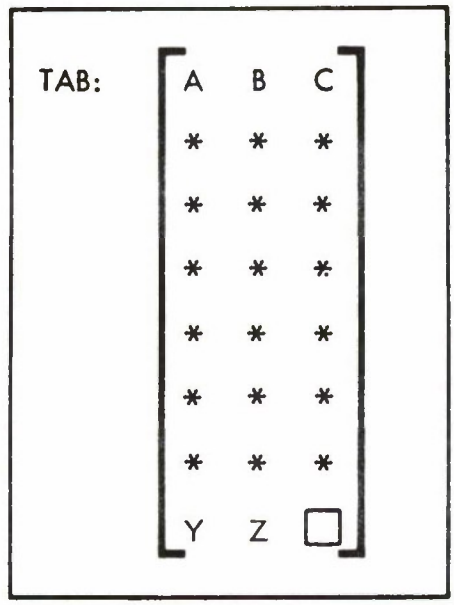


Figure 5. A Preset Character Array

The PL/I initialization feature has only two relatively minor annoyances. The first is that a multidimensional array must be initialized with a one-dimensional list of values; the second is that when a table is initialized, each item in it must be initialized with a separate list. Thus, the initial values for each entry cannot be grouped together in the natural way. But, again, these are minor problems.

Other Language Features

The list of requirements presented thus far may seem rather pedestrian and unexciting. Perhaps the most interesting part of it is what I have left off the list—and why.

As I mentioned before, I have left off my list of requirements all the ordinary features found in most high-level programming languages—things like machine independence, loops, and subroutines. These features are also required, of course, but are not worth discussing here.

Also, I have left out almost all of PL/I's powerful and exciting new features—things like the compile-time facilities, list processing, automatic and controlled storage allocation, recursive procedures, array and structure expressions, and so on. These features are all very useful, but are not essential requirements. The user decides whether he can afford them by trying to figure out how much they will add to the cost of acquiring, operating, and maintaining his compiler, versus how much they will save in programming costs.*

Of more interest, however, are the features missing from my set of requirements that you might reasonably expect to find in a list of command and control language needs. These other language features include: input/output and file processing, priority and interrupt processing, parallel processing, and machine-dependent processing.

There is certainly a need to do input/output and file processing in any command and control system, as well as the need to do priority and interrupt processing. Like it or not, some machine-dependent processing is usually required, and since most command and control systems involve at least an additional back-up computer, some kind of parallel processing is often necessary.

Except for the capability for machine-dependent processing, PL/I has all these features. But I do not include them among the essential requirements because there are perfectly acceptable ways of getting along without them in the language. Input/output and file processing, for example, may just as well be done by calling on library subroutines, and any necessary machine-dependent processes can also be library subroutines. The main thing that determines whether or not these capabilities are necessary in a command and control language is the operating philosophy adopted for the particular command and control system.

Operating Philosophy

For any command and control system, it is clear that a custom-built executive is necessary. Operating systems supplied by computer manufacturers are

*Experience with JOVIAL for command and control programming seems to indicate that even all the features of that relatively modest language are seldom warranted, much less the additional features of a language like PL/I.

simply too general-purpose and inefficient—even for a nonreal-time, command post information system. So the questions of operating philosophy, as shown in Figure 6, are the key issues in both command and control system design, and the choice of language features.

For any particular function, of course, the first question is whether or not it is really needed. I think the answer is "yes" for the functions I have just been discussing, so the next question is: Should these functions be performed centrally, in the executive, or in the individual subprograms?

In answering this question, the two major considerations are always efficiency of operation and ease of program maintenance. Efficiency of operation implies at least some degree of centralization, and ease of maintenance implies a need to minimize the interface between the executive and the individual subprograms of the system, so that one can be changed without necessarily having to change the other. This usually implies more centralization, which means these other language features are not necessary in writing the system's individual subprograms, because the executive performs these functions. With the help of relatively few machine-language subroutines, these features are even unnecessary in writing the executive.

It may be necessary, of course, for the individual subprograms to exercise some control over when and how the executive does these functions. This is done by having the subprogram leave a message for the executive, in a public table, or call an executive subroutine. But the programmer may just as well set the message in the table or call the subroutine himself, rather than have the compiler generate code to do it from some language feature, particularly since the former approach partly avoids the need to modify the compiler each time a change is made to the executive.

I should mention, however, that this argument applies mainly to the operational system—the application software—and not to the utility or support software. But a capability for priority and interrupt processing, for parallel processing, or even for machine-dependent processing is not usually needed in writing utility and support software, although input/output and file processing capabilities are required, if they can be implemented efficiently enough for production usage.

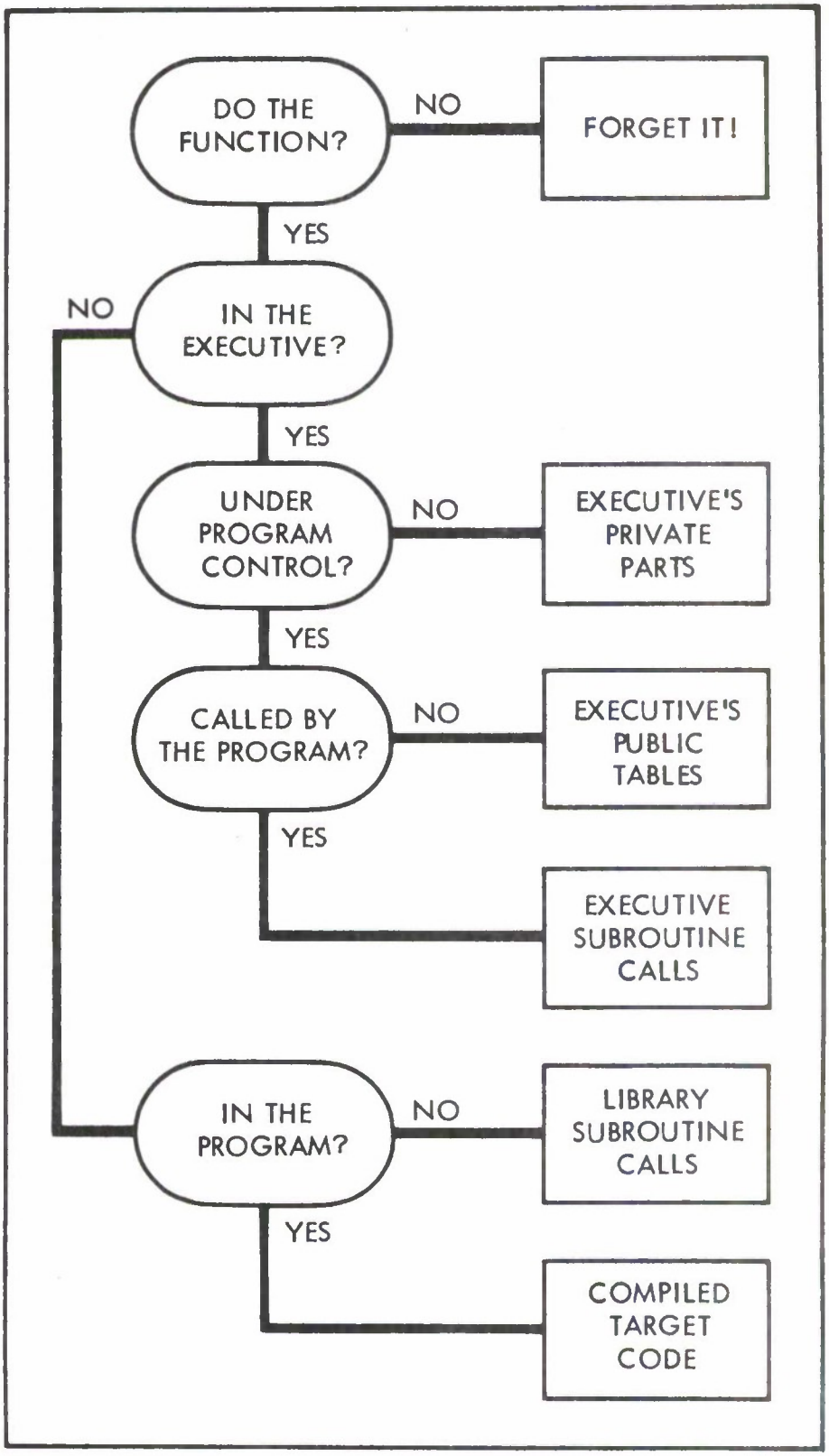


Figure 6. Questions of Operating Philosophy

THE UTILITY OF PL/I FOR COMMAND AND CONTROL PROGRAMMING

Command and Control Processor Needs

In addition to the programming language requirements I have been discussing, the command and control application area imposes fairly severe requirements on the language processor. It must be "compool" sensitive, it must interface with various executives, it must produce reliable, efficient target code, and it must be economical of implementation, operation, and maintenance. (And perhaps even courteous, brave, clean, and reverent.) Incidentally, these requirements are harder for a compiler to meet than an assembler, and they have considerable impact on the utility of PL/I for command and control programming, because they can be met only by picking a specialized command and control subset of PL/I, and custom-building a compiler for it. (The F-level compiler, for example, would be almost useless for command and control programming.) There is no question as to the feasibility of doing this—it is undoubtedly a practical approach.

Compool Sensitivity

To get down to specific requirements we must ask: What is a compool, and why does a compiler need to be sensitive to it? A compool (which is an abbreviation for "communication pool") is just a central file of data descriptions—a dictionary of data names—that are common to a set of programs. By referencing a compool, a generalized utility program, like a compiler, can translate back and forth between the names used by the programmer and the internal address codes used by the computer, and also between the external and internal representations of data values.

This is a vital function, but it is the centralization that is the key part of the compool concept, because it means each programmer can use the same dictionary, so they do not have to spend all their time trying to coordinate definitions. When a change has to be made, it can often be made once, in the compool, thus eliminating the need to look through numerous separate programs and make thousands of individual changes.

PL/I has two very useful language features that may sometimes be combined to make a compool unnecessary. One is the include statement, which allows the programmer to insert a string of PL/I text from a library file into his program. The other is the external attribute, which allows him to declare data elements that are common to more than one program. By including external declarations into a program, the programmer can get almost all the power and convenience of a compool. But there are some differences. There

is usually information in a compool that cannot be expressed in a PL/I text—for example, auxiliary and alternative storage locations. Also, a compool is a binary file, not a textual file, so it is much more compact, and this makes searching and maintaining it much faster. For a system with a very large amount of precisely allocated common data, I would say that you still need a compool.

Executive Interfaces

A compiler for command and control applications must necessarily interface with several executives. It must produce application programs that run under the operational executive. It must itself run under the utility executive, and if it is used for utility and support programming, it must produce programs that run under the utility and support executives.

In some cases, a single executive may suffice for all three systems, which naturally nullifies this requirement. But, while utility and support programs can often be run under the same executive, operational requirements usually dictate a special-purpose operational executive.

Reliable, Efficient Target Code

It is especially vital that a command and control compiler produce reliable target code because of the exceptional difficulty typically encountered in debugging on-line, real-time systems such as command and control systems. Anything that adds to this difficulty, like an undebugged compiler, must be avoided, or at least taken into account.

So far as efficient target code is concerned, there may be extreme cases where the computer is so big and fast in relation to the job that just about any code will do, or where the computer is so slow and small that it cannot solve the problem no matter how well it is coded. However, compilers <u>can</u> be built that turn out code that is good enough for most command and control systems. Since the computational requirements for most systems are almost always underestimated, it is typically just barely possible to do this. Thus, efficient target code is often the most stringent requirement, to which everything else—such as compiling speed and nonessential language features—must be sacrificed.

Economy

Finally, there is a command and control compiler requirement for economy of implementation, operation, and maintenance. This requirement is imposed not so much by the tyranny of a fixed budget as by the tyranny of a fixed schedule. A compiler for a "big" language like PL/I takes a lot of time to build—as IBM has found out—and since a custom-built rather than an off-the-shelf compiler is usually required, this can have a great impact on the overall schedule for a system. A big compiler for a big language is almost necessarily a slow compiler. Although economy of operation is always desirable, when there are, say, 100 programmers frantically trying to debug their programs at the same time, a slow compiler can mean they get many fewer shots at the computer—even a time-shared computer—and this can have a great impact on a schedule. Economy of maintenance does not affect schedules as much as it does budgets, but it is still a worthwhile goal.

All the requirements I have been discussing can be met by a good compiler for a suitable subset of PL/I. But I would caution against trying to include too much in that subset—aside from the features that are obviously needed in any programming language, and the features I mentioned earlier that are needed for command and control. Again, any additional features should be implemented only if they will save time or money.

PL/I's Command and Control Prospects

My comments on PL/I's prospects in the command and control field will probably be unsatisfactory because they boil down to this: Your guess is as good as mine—at least for the long term. But for the short term, say the next three or four years, I think PL/I's prospects are surprisingly poor for becoming the standard command and control programming language. I say surprisingly, because the language (or rather a dialect of it) is clearly suitable for command and control, as I hope I have made clear, and because a great many people, supporters and detractors both, think it is the wave of the future, and this by itself generates considerable demand for its use.

There are many reasons why this demand will not soon be widely satisfied in command and control. One reason is simply that the Air Force has only recently established JOVIAL as the standard computer programming language for Air Force command and control systems, and I do not anticipate an early change to this policy. The Navy adopted JOVIAL several years ago as the

standard language for its strategic (i.e., shore-based) command and control systems, though it uses CS-1 for programming its tactical (i.e., shipborne) systems. Only in the Army, which seemed until recently on the verge of choosing JOVIAL as its standard command and control language, are there any prospects for the use of PL/I. Dialects of PL/I are being developed for the Sentinel anti-ballistic-missile system and for the Tacfire artillery fire control system. In addition, there are signs of reluctance, at certain high levels within the Department of Defense, to encourage the use of any programming languages but COBOL and JOVIAL.

Another factor that will slow the acceptance of PL/I for command and control programming is the lack of an established and suitable dialect. Nobody has implemented or even announced a command and control dialect of PL/I yet. The Army's projects are evidently still under development. An IBM project to develop a military subset of PL/I is still at an early stage, and a small SDC project has only recently begun to look into requirements.

For the long term, however, if PL/I ever does become the universal programming language, supplanting FORTRAN and COBOL for scientific and commercial applications, then it would clearly make a lot of sense to use it for military applications too. Some say this is sure to happen. Others say PL/I will remain a toy, like ALGOL. At any rate, I would not recommend changing the Air Force command and control programming language standard from JOVIAL to PL/I until somebody can demonstrate that a lot of time or money can be saved by doing it.

References

1) "IBM System/360 Operating System, PL/I Language Specifications," IBM Form No. C28-6571-4 (1967).

2) "Standard Computer Programming Language for Air Force Command and Control Systems," Air Force Manual No. 100-24 (15 June 1967).

3) Shaw, C.J., "Computer Programming and Command and Control," System Development Corporation Report No. TM-2857 (25 April 1966).

4) Shaw, C.J., "On Declaring Arbitrarily Coded Alphabets," Communications of the ACM, Vol. 7, No. 5 (May 1964), pages 288-290.

5) Engdahl, S.L., "Principles of Program Design for Large-Scale Command and Control Systems and Associated Support Systems," System Development Corporation Report No. TM-3396/000/01 (1 May 1967).

Questions and Answers

Question: In reference to your comments on the prospects of PL/I in general, to what extent is the future of PL/I tied to the 360 computer? Is it conceivable that the 360 would be operable without PL/I?

Answer: Lots of people are using 360s who never heard of PL/I.

Q: Do you envision it as a possibility for the future that the 360 would be usable to a large extent without PL/I?

A: I don't see why not. After all, there are FORTRAN and COBOL and ALGOL. And if enough people decide it's not worth making the switch to PL/I, why then they won't. You presumably are going to have to convince them that they're going to gain more than, say, 20% reduction in programming costs by switching to PL/I—otherwise it's not worth the effort. There's an awful lot of inertia involved.

Q: Going back to that slide where you have the replication factor—(18) (1) '*'—what did you say about the use of (1) which avoided replication of the asterisk?

A: Well, if I'd left out the (1) I would have specified a single character string composed of 18 asterisks. This would be a single value, and would serve to initialize one of the elements of the array, where in fact I wanted to initialize 18 elements of the array with a single asterisk; so I had to prefix the asterisk with a repetition factor of 1 to indicate that I wanted a 1-character string. This is just a coding trick, if you will, in PL/I.

Comment: The reason it's necessary is that if you do want a character string which contains 18 asterisks, you can specify it without explicitly stating 18 asterisks by saying (18) '*'.

FIRST PANEL DISCUSSION

Participants

Frederick P. Brooks, Jr., University of North Carolina
Vilas D. Henderson (Moderator), Logicon
Mary D. Lasky, The Johns Hopkins University
Robert E. Rosin, Yale University
Christopher J. Shaw, System Development Corporation

Henderson: Bob Rosin this morning raised the point that I think was also brought out quite well by Chris Shaw's talk this afternoon: there is going to be a continual problem regarding who controls the language PL/I and what happens in connection with the selection of subsets and so forth. Let's start this off by hearing from Bob to get his thoughts in this area.

Rosin: It should first of all be clear, as I suspect that most of you know, that PL/I originally evolved as "NPL" out of a joint SHARE/IBM effort, three men from each organization. It later was digested by IBM and came out as an IBM tech report in December '64; this tech report was later revised into the -0 and eventually the -4 language specifications manual. Since it went into IBM—and I'm not questioning whether this is proper or not at this juncture—it has been controlled by IBM to the extent that they have built a very elaborate, and in many cases probably well-justified machinery for deciding what does go into the language, what doesn't go into the language, and in some cases—much more sensitive cases—what aspects of the language are candidates for review and perhaps change. Many people, I should point out, have implemented alternative subsets of PL/I, and in many cases, as Chris points out, alternative dialects. That is, they have not only constrained the language in some cases, but they've also modified it, in some cases quite considerably. We get into debates occasionally in SHARE, and occasionally just sitting around the campfire some nights, as to which one of these things—if any of them or if all of them—is really PL/I. It's clear, for example, that the F-level implementation is not PL/I; it's a subset. For example, list processing, which is now part of the language, is not at all implemented in the F-level compiler. IBM has implemented a subset, which is almost a proper subset, for the D-level compiler which is to run under the DOS system. There are other dialects which look like ALGOL with character strings, and so forth. There are other dialects which look like FORTRAN with semicolons. Some of you may have seen documents for some of these things.

It's clear to those of us working in SHARE that if we want to develop PL/I in SHARE, then we must cooperate with IBM and somehow work with this structure. Now I say again, if we want to work in SHARE we must do this; it's clear that this is the case because SHARE is not about to go out and build a PL/I compiler, nor is any relatively significant group in SHARE going to do this and distribute it. We have experiences (before my time in SHARE) with something called SOS which indicate that this may not be the way to go. If we as customers of IBM expect IBM to do it, we therefore must interface with their procedure. On the other hand, an awful lot of us have been getting up on soap boxes occasionally and talking about alternative definitions of various features of the language: in some cases rather significant, and in other cases rather impudent extensions to the language. I think the kinds of things Chris was talking about this afternoon were really quite pertinent. I grew up in an ALGOL 58 dialect, as is JOVIAL, where, indeed, data is data, and you can define identifiers of arbitrary data types to overlay one another in sometimes rather confusing, but in many cases rather powerful ways. Obviously, this would require a rather radical redefinition of what is PL/I, in terms of the concept of data in PL/I. Now, the question that arises is: If one does make this radical redefinition and comes out with something that has the flavor of PL/I in some sense and the flavor of, say, JOVIAL in another sense—although I will discuss in a little while a proposal that has the flavor of SNOBOL, which is really in some cases radically different—is this still PL/I, is it a PL/I dialect, is it a PL/I subset, or is it really a kitty with a different kind of stripes? IBM knows who controls the language. USASI has a committee that's trying to discover whether they're good enough to find out what PL/I is and define it. I'm not sure exactly where we're going.

Henderson: Fred, do you have anything you'd like to say about this?

Brooks: It seems to me the following lesson can be drawn from the COBOL experience. Control of the language means several things: one is policy decisions about features, and the other is the detailed technical working out of syntax and semantics. And that latter function is not a committee function; it is a one-man function. You need a uniform conceptual approach which is essentially the function of one mind. Now, I think it is perfectly possible to combine that technical work with policy decisions arrived

at by committee. I know this is possible because I've seen machine definition done that way, in which one man writes the manual with his own pencil but all kinds of policy decisions as to what priorities and features would be designed are made by a group.

Shaw: That's how JOVIAL was designed. I wrote the manual and we had a large committee make the policy decisions.

Rosin: There are some people who feel that PL/I is too big for one man.

Shaw: It isn't.

Brooks: It isn't. No. Ask George Radin.

Rosin: I'm not sure today what position he would take. It is rough; it's a lot more difficult than it used to be, although I agree with you that the technical details....

Shaw: The reason it is so difficult right now, Bob, is that the PL/I manual has 15 different authors; each one has written a section of it without reading what the rest of the people have written.

Rosin: Let me clarify for the group, in case they're not aware, what the procedure looks like inside of IBM, as I understand it. There is a Language Control Board and there is a Language Review Board; I'm never exactly sure which is which, or which does what. The Language Control Board controls what is going on in the language by making sure that implementations maintain the letter of the law and extensions somehow bear in some consistent way upon what already exists. The Language Review Board is that group which is responsible for considering extensions and changes. Now that obviously is not one man. There is, however, a language manager somewhere in IBM who is responsible for this.

Shaw: But he doesn't write documents.

Rosin: That's true.

Shaw: He should.

Brooks: He should; there has to be a person, somewhere. We have to distinguish between this essentially technical fleshing-out function and

the policy-making function. I think the second thing to be learned from the COBOL experience is that policy-making decisions should reflect responsibility. Anybody who has made a significant commitment, as a user or as an implementer, probably should have some mechanism for having his voice heard. But anybody who is merely curious, and has not made a significant commitment on his own part, ought to be rigorously excluded. Otherwise you get too many dabblers who have nothing to lose; I think we've seen that process on character code standardization.

Henderson: Mary, do you want to say anything about this area?

Lasky: From working with the SHARE committee on it, I know what goes through when the SHARE project would like to put a change in the language, and this change doesn't agree with GUIDE, and then IBM certainly doesn't like it—it goes around and around. The work that goes on to get something into the language is certainly not trivial, not easy, and not fast—and in a way this may be good. I think the language control could be taken outside of the hands of IBM and perhaps given to some body like USASI so that they could form a standardized language that would be the PL/I. If you wanted to go outside of it and do a subset or something, that would be fine, but you would have a guideline for doing variations. It will certainly be a benefit to the other manufacturers who haven't gone forth with vigor at this time in what they're doing with it; if some grander body than IBM were holding the strings to the language, then they could also have a voice on whether something gets changed in it. So I think that this has a benefit. Being from a scientific installation and also from a university where many of our programs come from other installations, I have misgivings about a lot of subsets creeping up, with their incompatibilities. It would be nice to have something that's clean enough that I could send my PL/I program to another installation and have it run without a lot of modifications. If you get too many little bits and pieces growing up because people find that one feature doesn't satisfy them, so they add another little feature here and there, then programs are no longer compatible. You have this with FORTRAN now, and have gotten into trouble. I think that unless you guard definitely against this with PL/I, you're going to get into the same bind. I don't know what the answer is, or what's going to happen with USASI. I know they are looking at it now and deciding how the wording should be about whether they will go ahead and standardize it or not.

Henderson: Chris, how about you?

Shaw: This business about "What is PL/I?" is, in fact, a very difficult question to answer because as yet there has been no definitive compiler for the language. Without a definitive compiler for the language, a lot of the language features that are as yet unimplemented are open to more or less serious questions as to their practicality. We've had similar problems in nomenclature within SDC concerning JOVIAL. We've gone over these problems and come up with various solutions to them over the years. At one time we were going the central committee route, where there was a central committee, and they, by God, said what JOVIAL was. This didn't work well for a number of reasons. The approach we are taking now is actually an old approach. It is the approach used by the implementers of NELIAC toward defining what they meant by NELIAC: that approach is simply to specify the basic subset of the language. This permits a reasonably definitive compiler to be written for this basic subset, so that if questions aren't answered in the manual as to what's in the language, you can look at the compiler. And there will always be questions that the manual won't answer. Now in standardizing JOVIAL within SDC, what we have done is standardize our nomenclature for JOVIAL. So we say that any compiler that calls itself a JOVIAL compiler has to accept Basic JOVIAL, and produce the same results—essentially the same results—as our definitive compiler does for the same input program. We also have one or two other official dialects of the language; these are necessarily supersets of Basic JOVIAL. Anybody can build a JOVIAL compiler, but if he is going to call it a JOVIAL compiler it has to accept Basic JOVIAL. He can add on this basic framework anything he wants and still call it a JOVIAL compiler, the only restriction being that it has to be compatible with the only official JOVIAL dialect, namely, J3. I think that this is probably a pretty good way toward standardizing what you mean when you use the work "PL/I," and this is essentially all that we really want to do. We want to define PL/I. And I think this is a much better way to go than the COBOL route, where they have a standard definitional mechanism for COBOL that allows you 7000 different dialects of the language in combinations of various modules. I think the USASI standard FORTRANs—the basic FORTRAN and the full FORTRAN—could be used in this fashion, the same way that we use Basic JOVIAL and J3 JOVIAL. And I think this is a very good approach; however, I don't see the industry approaching this solution for PL/I at a particularly fast

rate, I do know there is within IBM a very strong effort to provide a rigorous formal definition of the language.

Rosin: I made a few notes on Chris's comments. First of all, the comment about standards: from a practical point of view—this is a cynical practical point of view, I must admit—standards are supposed to provide a groundwork upon which things are based, and apparently this is the case with JOVIAL. One finds, however, that when he confronts a manufacturer with a standard and then asks for something to be produced according to that standard, the standard then becomes the upper bound as well as the lower bound. This is a little frightening. This is not always the case, but the extensions above the standard are very frequently minimal.

Shaw: If you want more, go to them with more money.

Rosin: Oh right, right. I'm now speaking as a small user in a large community. It's difficult. I think one way out of this may be the concept of an extensible compiler and extensible language, where there is a core PL/I processor which includes the ability for self-definition and self-extension. (This is something we talked about, and I must confess we've talked about this in the SHARE PL/I project, generating a lot more heat than light—probably infinitely more heat than light.) Now, the PL/I compile-time facilities, as they exist now, to be very blunt are in my impression a mockery of this concept. I think the people who are responsible for this in PL/I were aware of it then and are aware of it now. This is a very difficult problem. There have been several papers in this area recently; they've all been good, and I think they've presented a challenge to the compiler-writing community. I suspect PL/I is an area that might profit. Let me just make a couple of other comments.

Brooks: Are you going to leave that subject?

Rosin: I'll let you come back to it.... OK, go ahead.

Brooks: It seems to me that the principal reason for having a standard is so that you can trade programs; and if you're going to be able to trade programs, you have to have more than a minimum standard. That is, you have to have a standard standard, one that defines the upper bound as well as the lower bound. This

is a degree of rigor which no compiler designer or language designer has been willing to accept. We have finally got to the point that we know how to do it with machines and have indeed been able to muster enough discipline to do it. Not so with languages. Perhaps this reflects the ten years of extra maturity in one field compared to the other.

Shaw: One of the problems with language design is that a lot of the theoretical work being done in this area is, in my mind, equivalent to writing compilers for Turing machines. You can do it—you can presumably write an ALGOL compiler for a universal Turing machine that defines ALGOL for you very nicely—but so what?

Brooks: Let me give a more concrete illustration of what I mean. When we built STRETCH, every time anybody could think of something that would make the machine look like it was faster, we flung it in.

Shaw: That's how they designed PL/I!

Brooks: And the result was a machine that was either baroque or rococo depending upon....

Shaw: That's PL/I!

Brooks: The thing we learned in looking at the 360 line was that the requirement of looking at the future, which the big machines clearly imposed on the small ones, benefitted the small machines. More surprisingly, the requirement of close attention to cost, which the little machines imposed upon the big ones, was good for the big machines. And the requirement of downward compatibility was the best thing that the Models 65 and 75 ever saw because it kept them disciplined. I think the same kind of discipline imposed on languages—to say, "All right, you build to this standard, and you do not build above this standard"—would save an awful lot of garbage getting in.

Rosin: I agree with you wholeheartedly. The essence of my point is that if the standard PL/I included an extension facility, then one might be half-way home.

Shaw: A usable extension facility, one that allows you to do just about any translation your heart desires.

Rosin: To be able to define data types, to be able to define operators, statement types, and to generate code.

Brooks: In building an extension facility one has to be careful to preserve both machine independence and implementation independence. If your extension facility says, "Boy, hold, I can change my tables provided your compiler is built using a certain table structure," you have lost rather than gained.

Rosin: I might say that the "flavor" of ALGOL 58 that was used in Michigan with the MAD compiler does have a rudimentary extension facility in it, and it turns out to be pretty powerful at times. That is, you can indeed define new operators, and you can do this without redefining the language. This is a powerful tool, both for the experimenter and for the guy who has a production job.

Let me just offer these other two points, and I'm sure this will get somebody's hackles up. There are two forces operating here which are very interesting. In some ways they appear to reflect the production community as opposed to the research community—those people with problems to solve as opposed, perhaps, to those people who invent problems. There is first of all, the requirement for stability; IBM sees this very, very deeply. Again I say this from my working with people in IBM on the policy level of PL/I. If PL/I changes too much, PL/I will not be adopted by anyone. They insist on this, and this is one reason for standards, one reason that companies like Univac, Honeywell, and so forth, are adopting a fairly heavy-handed wait-and-see attitude. On the other hand, in a university research-oriented community, change is the status quo; and it doesn't make much difference to me in teaching my courses if PL/I changes much from year to year, because if I do a good job in teaching, my students will be able to recognize the change and adapt to it without any trouble at all. If they couldn't do this, then they failed the original course, and I failed in teaching the original course. Another way of looking at this dichotomy of stability and change is perhaps "the market" versus "excellence"—where excellence is defined by something which is rather dynamic. Now that's sort of an ivory tower picture, and to avoid Chris's sin, I don't want to really say....

Shaw: Which one?

Rosin: The dichotomy problem—not your sin, the sin you're saving us from. I don't believe there's really a dichotomy; it's obviously a continuum and it's probably multidimensional. Let me just ask a question of the panel. It's a serious question; it's a question that comes up, for example, in SHARE meetings; and I got badly trounced up in San Francisco for proposing a resolution which would, in effect, meet some of these ends: What would happen to PL/I, in terms of acceptance, in terms of a controlled environment, in terms of your teaching, and, Mary, in the attitudes of your scientific programmers, if somebody who had capabilities to program and support compilers turned around tomorrow and delivered a PL/II compiler, where PL/II was substantially different from PL/I? What would be the effect on this group sitting here?

Shaw: I'll tell you what did happen! At SDC this last year we have been designing a programming language, a fairly powerful language for the spaceborne application area. The requirements for this language encompassed all the things that PL/I has and a half a dozen other things. We came up with a language spec for a language like this—quite a bit different from JOVIAL or PL/I or what have you; it was a little bit like CPL, and a little like the Iverson language. Our next job was to JOVIALize it because the Department of Defense didn't want it.

Brooks: I'll tell you another example. Mary said most of the code they are running on the 94 is written in FORTRAN II.

Lasky: For one thing, as Bob said, we're one of the few installations in the world that took over the SHARE Operating System. We have done a great deal of our own work using SOS, and have added other language processors onto it, and have gone through the experience and expense of maintaining our own software. With only five or six people in the systems group, we don't want to do it again—and we can't afford to do it again as the systems become larger and larger and you get more powerful machines.

Brooks: Yes, I think we see the same thing happening on three levels. As an individual I raced out into the world waving a flaming sword, all for change and down with stability—that was in 1956. I am now much more in favor of stability. I think that as individual installations, the computer users have also been continually re-evaluating this trade-off in favor of higher and higher emphasis

on having things stay still long enough to get some work done. And I think that as a profession and a discipline we are placing higher values on stability. And so I think at all three levels—individually, by installation, and as a discipline—we are learning that the true inertial forces are much greater than we thought they would be. The thing that's happening is that people are building parameteral programming systems in which you build something, and then you build more on top of it, more and more and more on top, and today we are much more talking about and building whole systems rather than lots of little isolated programs. The conversion problems are just orders of magnitude different between these, so I think we're going to see the whole works stabilized more and more.

Rosin: Fred, one comment I heard about PL/I—to which several people have attributed some of its lesser positive qualities—is the fact that somebody, maybe a committee, decided that PL/I should be ready about the same time the 360 was ready, and that it probably needed a year or two more of thinking and experimental compiler building and so forth before it was cast upon the world.

Brooks: That doesn't correspond to my recollection. PL/I was developed independently of System/360 and its timing was independent so far as I know.

Instrument noise transferred to the tape precluded transcription beyond this point, which occurred about midway in the panel discussion.

STRINGS AND ARRAYS IN PL/I

Robert F. Rosin
Associate Professor,
Engineering and Applied Sciences
Yale University
New Haven, Connecticut

Author's Note

The following pair of papers dealing with the facilities provided for manipulation of array and string data in PL/I are the result of many months of examination and use of these features of the language. It might appear that I am highly critical of these aspects of PL/I, but that is not my intention. PL/I provides function and notation for manipulation of these forms of data in a manner far superior to that available in any other widely accepted general purpose language. It is my intention to illustrate these features and then to highlight a series of language extensions which would serve to continue the spirit of excellence upon which the language is already based.

ARRAYS IN PL/I

Introduction

As it stands, PL/I provides a far more general array manipulation facility than does any other widely accepted general purpose language (Reference 1). On the other hand, an examination of what is provided reveals some rather unfortunate omissions when considered in the light of completeness and symmetry. Any serious attempt to use the language makes several of these cases painfully obvious. This discussion will begin with a brief examination of the array manipulation facilities defined in PL/I and will conclude with a set of suggested generalizations and extensions. Strings are often considered a special case of arrays; however, they are treated separately in another paper by the same author.

Existing Facilities

The defined (and implemented) array facilities in PL/I include the following:

1) Declaration of arrays of arbitrary dimensionality with arbitrary extents in each dimensionality

2) Declaration of an initial value for a given array

3) The ability to allocate and free storage assigned to an entire array but not its constituent elements

4) Element-by-element arithmetic on arrays of mutually consistent dimensionality

5) Arithmetic operations combining arrays and scalars

6) Passing of arrays as arguments to procedures

7) Declaration of other than 1-to-1 mappings between arrays: iSUB notation

8) A set of built-in generic functions for performing certain mathematical manipulations and for determining extents and bounds of arrays

9) Consecutive subscription and array cross-sections.

It is clear that these facilities constitute an advance and generalization of array manipulation, given that languages such as FORTRAN support, at least in part, only items 1, 2, 6, and 9.

However, one simple example is adequate to point out the nature of the existing deficiencies. Consider the task of writing a procedure to perform an in-core sort of numeric data. This can be accomplished with relative ease in PL/I, except for the following unfortunate limitations. First of all, one would prefer to write SORT in such a way that it could be used as follows:

```
X = SORT(Y);
```

However, one discovers that PL/I does not allow array valued procedures in any form.

This disappointment is supplanted by another when one chooses to redefine the procedure so that:

```
CALL SORT(Y,X);
```

is used in the calling program. In this case one is quickly confronted with the fact that if the target array, X, does not agree in all attributes save dimensionality (which can be arbitrary) with the dummy parameter in the procedure definition, then the sort will take place but the value of X will not be changed. PL/I does not convert values associated with array parameters except when passed into the procedure.

Having decided, therefore, to assure the absolute consistency of array attributes in a program, one might choose to use data-directed input to initialize the array. Thus:

```
PROG: PROC OPTIONS (MAIN); DCL (X, Y)(30); DCL SORT ENTRY ((*), (*));

LOOP: GET DATA; CALL SORT (X, Y);
      PUT DATA X; GOTO LOOP;
      END PROG;
```

One is confronted with the prospect of supplying the data to be sorted in the following way:

```
Y(1) = 34.2, Y(2) = 56, Y(3) = 77, etc.
```

That is, unlike in the INITIAL attribute where one can say:

```
DCL Y (5) INITIAL (34.2, 56, 77, etc.);
```

the user is forced to use a dummy assignment statement for each element of the array rather than for the whole array. One would prefer to be able to say something like:

```
Y = (34.2, 56, 77, etc.);
```

These comments may appear to be "nit-picking"; but the author has, unfortunately, been confronted with exactly this situation in attempting to provide and use a sort facility in his installation subroutine library. It is clear that these and other defects can be corrected by providing additional syntax, but it is not always obvious which of several alternatives is best. Furthermore, early IBM implementation strategies appear to rule out reasonable interpretation of a few of the following suggested changes; and the generality implied by several might lead to inefficiencies in any implementation. Therefore, this area of PL/I requires serious study in order to meet the two objectives of notational ease and computational efficiency, which can be in conflict.

Suggested Extensions

The suggestions which are enumerated here fall into the following two categories: symmetry of PL/I functions and incompleteness of facilities. In addition, some of these changes are obviously acceptable and others are quite controversial. No attempt is made to place any proposed extension into any of these categories.

1) A single notation should be defined for array constants, and this notation should be universally applicable in declarations, expressions, and I/O. For example:

```
DCL X (3) INITIAL (1,2,3); DCL (A,B)(3);
A = B+(1,2,3);
PUT LIST (1,2,3);
GET DATA X; with the data card containing X = (1,2,3);
```

The use of nested parentheses should be allowed for definition of constants of dimensionality greater than one, e.g.:

```
DCL Y (2,2) INITIAL ((1,2,),(3,4));
```

and incomplete rows and columns (etc.) should be filled with zeros.

2) Array expressions should result in array temporaries and not the DO loop interpretation currently provided, e.g.:

```
DCL (A, B) (3, 3);
A = A/A (1, 1); A = B;
```

should behave exactly like:

```
B = A/A (1, 1); A = B;
```

rather than setting the first element of A to 1 and effecting no other change. A facility for array temporaries already exists in any PL/I implementation to allow for conversion of array parameters when passed to procedures, and it is not clear that this reinterpretation would necessarily be less efficient than the current approach. Other examples of the positive effects of this change appear in other proposed extensions.

3) Array subscription should be expressible in one or two ways, and this notation should be allowed in all contexts. For example:

```
DCL A (10), B (3);
B = A (4:6); X = (B+A(1:3)) (2);
```

where the latter extracts the second element of the sum of B added to the first three elements of A. In the case:

```
B = A (6:4);
```

the 1st element of B should contain the 6th element of A, the 2nd element of B should contain the 5th element of A, etc. Again, the use of array temporaries would provide quite readily for this generalization. It is not clear that:

```
(array expression) (subscript expression)
```

is syntactically unambiguous, and an explicit subscription operator might have to be provided for this case.

In I/O statements one is allowed to use DO notation for element selection, and there is no reason not to provide such a facility for expressions and declarations. E.g.:

```
DCL A (10) INITIAL (A(I) = (3, 4, 5) DO I = 1 TO 5 BY 2);
DCL X (3); X = (A(I) DO I = 1 TO 5 BY 2);
```

4) Array valued functions should be allowed. Consider:

```
DCL SORT ENTRY ((*)) RETURNS (*);
SORT: PROC ((*))(*);
```

Again the question of interpretation is important, and the use of array temporaries rather than the current definition appears to solve the problem. One should be able to write:

```
DCL X (4) ; DCL Y (10);
X = (SORT(Y)) (5:2) + (1, 2, 3, 4);
```

and if Y contains 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 in any order, then X should contain 6, 6, 6, 6.

5) In addition to the ability to allocate entire arrays, it is reasonable to consider providing a facility for reallocation of portions of an existing array, including adding and deleting rows, columns, etc. and reallocation of existing data for use by alternative accessing schemes (by other than overlay defining). The former might well be inefficient, requiring reallocation of the existing array, transferring of the selected elements, and deletion of the old copy. The second scheme would require only the respecification of the parameters passed to the accessing function, which supports the subscription notation and is currently available in at least one dialect of ALGOL '58 (MAD).

6) Although the ability to define generic functions provides a portion of the facility desired (and should not be discarded), the ability to obtain the attribute values of any identifier, either internal, external, or parameter, would be of great value. Consider, for example, the ability to select a differing code in a sort procedure depending on the nature of the data to be sorted—REAL, COMPLEX, or CHARACTER. This would accomplish a generalization of the currently provided capability for the programmer to determine the extents and bounds of arrays at object time.

Several alternative schemes come to mind. For example, the language might support a separate built-in function for each possible attribute which would return a character string, perhaps empty, containing the appropriate attribute value. More general would be a single built-in

function which would return a vector of the attribute values which could then be manipulated by the object program. This latter scheme would allow for extension of the attributes which the language supports without necessarily obsoleting previously written programs.

Conclusions

It is clear that PL/I provides facilities for manipulation of arrays which supersede those of any general purpose language of wide acceptance. It is equally clear that these facilities can be made more symmetric and complete, and remain within the spirit of the PL/I language. One would hope that the suggestions in this brief paper would be considered by the programming community, and especially by those responsible for furthering the development of PL/I.

References

1) IBM Corp., "IBM System/360 Operating System: PL/I F-level Reference Manual," Form No. C28-8201.

CHARACTER STRINGS IN GENERAL PURPOSE PROCEDURAL LANGUAGES

Abstract

It is pointed out that PL/I is the only widely accepted general purpose language which supports the character string data type to the same functional extent as do SNOBOL and COMIT. The PL/I facilities are briefly described and then a set of notational (but not functional) changes are proposed and discussed. The conclusions drawn in this paper can be readily applied to the inclusion of character strings in any general purpose languages.

Introduction

Although there have existed several languages whose primary data type is the character string (abbreviated henceforth as CS), notably SNOBOL (References 1 and 2) and COMIT (Reference 3), general purpose procedural languages are seldom able to support fully this most useful form of data. The complex pattern matching and substitution facilities of the two languages mentioned are obtained only through extremely complex and often syntactically "illegal" constructions when considered in the context of FORTRAN, COBOL, and the several ALGOL dialects. Furthermore, processors for these languages seldom support the particular form of dynamic storage allocation needed for varying length strings as outlined by Madnick (Reference 4).

The one notable exception in current use on a widely accepted basis is PL/I, which offers both the fixed and the varying length CS and a set of operators, functions, and implicitly called conversion routines capable of very general CS manipulation (Reference 5). These same facilities are also applicable to bit strings, a fact with which we shall not be concerned, but which does not invalidate any of the discussion offered in this paper.

It is the purpose of this paper to present a brief summary of the existing facilities in PL/I for CS manipulation, both in terms of form and function. Following this we will offer a set of possible language changes which provide the same function, but in a far more convenient notation. Finally the validity of these changes will be discussed in terms of a set of critical statements which apply to both the notation proposed and implementation of this notation extension in a compiler.

Although directed to PL/I, the remarks concerning CS in this paper are relevant to any attempt to provide a comprehensive CS facility in a general purpose procedural language. Notational simplicity is a primary reason that these languages are used. To provide a new function by offering notation which is awkward, although concise and complete, is to defeat one goal of defining such a language in the first place. Nevertheless, it is the author's contention that the CS facilities included in PL/I constitute a marked improvement over those offered by any other general purpose language of wide interest; and its developers should be recognized for this advance.

The Existing PL/I Facilities

The facilities afforded by PL/I for CS manipulation are summarized here. Of course, one cannot neglect the additional facilities included in PL/I which are also useful for general algorithm construction and for CS as well as other data types.

PL/I provides fixed and varying length strings. In both cases, the maximum length of the string must be stated at the time that the string is declared. A CS can be a member of any larger data aggregate, array or structure, and can appear in based storage. CS constants consist of any string of characters delimited at both ends by a pair of apostrophes, with an adjacent pair of included apostrophes representing a single occurrence of that character. A constant replication factor can be applied to such a CS constant by enclosing the replication factor between a pair of parentheses immediately preceding the first apostrophe.

CS subscription is accomplished by the built-in function SUBSTR, which has a CS as its value. SUBSTR (X, I, N) returns the substring of X from the Ith character, N characters long. Exceeding the limits of X currently results in a truncated value, but more recent proposals have suggested that a STRINGSIZE condition will be raised instead. If SUBSTR is called with only two arguments, then the entire CS starting from the Ith character is returned.

SUBSTR can also be used in the sets position in an assignment statement, allowing the programmer to assign a value to a portion of some existing string. The length of the CS being assigned is adjusted to match the target of the assignment. For example,

```
SUBSTR(X, 4, 3) = 'ABCD';
```

replaces the 4th, 5th, and 6th characters of X with 'ABC'.

PL/I also provides the full set of relational operators and a very complete set of implicitly invoked conversion procedures, allowing the use of CS data in more traditional contexts. Also included is an operator, ||, for denoting concatenation.*

Finally, scanning and pattern matching are supported through another built-in function INDEX, which returns an integer as its value. INDEX(X, Y) returns the index of the first character of the left-most substring of X which matches the CS Y. If no such substring exists, then INDEX returns the value 0 (zero).

The examples in the appendix illustrate a few simple cases of the use of these facilities along with their counterparts using the notation in the following section of this paper.

Proposed Language Changes

This proposal and the appended comments have resulted from many months of thought and speculation about the CS facilities in PL/I. To be sure, the existing language tools are complete in the sense that one can program any reasonable scan, replacement, extraction, etc. However, it has been considered by some that a more natural set of CS operations should be considered for this language so that reasonable applications could be approached in a straightforward way.

Much of the flavor of this scheme will appear to be reminiscent of SNOBOL, and not without cause. SNOBOL has been used with great satisfaction by the author and his students, and it is considered by many to be the most successful of all attempts to provide a special purpose programming tool built around the CS data type.

---

*The precedence of this operator has recently been changed. Previously, it had lower precedence than the logical connectives & and |, which themselves were below the relational operators. The change places || immediately above the relationals. Therefore, the sequence

X = A||B; Y = C||D; IF A||B = C||D THEN...

now groups expressions in a "more natural way" without the use of parentheses.

On the other hand, every effort has been made to preserve the style and spirit of PL/I. Indeed, the one concept which might tend to cause some consternation is the generality with which the left-hand-side expression, or pseudovariable, has been used. This becomes a very powerful tool when freed from the bounds of functional notation as imposed by SUBSTR. But this proposal does not include non-PL/I concepts, such as the arbitrary use of assignment within expressions which does predominate in SNOBOL.

Special problems associated with BIT as opposed to CHARACTER have not been considered in depth, but no difficulties have appeared so far. Examples of the use of this extended notation appear in the appendix.

The proposal constitutes the following 12 items:

1) The default attribute for CS should be VARYING, with the option to declare FIXED. The maximum length, if specified, would serve as a guide to the compiler, but is optional.

2) SUBSTR(X(A), I, J) should be replaced by X(A:I...J+I-1) or some similar notation.* It follows that:

   X(A:I) ≡ X(A:I...I)

   X(:I) ≡ X(:I...I)

   X(A:...J) ≡ X(A:1...J)

   X(A:I...) ≡ X(A:I...LENGTH(X(A)))

   In an expression such as X(:I...J) if 1 > J then the value of the expression is the specified substring with the characters appearing in reverse order. For example,

   X = 'ABCDEFG'; Y = X(:5...3);

   results in Y contain 'EDC', and

   X = X(:LENGTH(X)...1);

   reverses the characters in X.

---

* It has been proposed that the following alternative be adopted: X(A, I:J+I-1).

The implication of such a construction in the ultimate assignment position should be identical to X(:J...I), that is, without reversal.

3) Five new operators should be defined. Their purpose is to scan strings, left to right, and return a string temporary.

X UPTO Y returns X(:1...INDEX(X,Y)+LENGTH(Y)-1)

X BEFORE Y returns X(:1...INDEX(X,Y)-1)

X AFTER Y returns X(:INDEX(X,Y)+LENGTH(Y)...)

X FROM Y returns X(:INDEX(X,Y)...)

Y IN X returns X(:INDEX(X,Y)...INDEX(X,Y)+LENGTH(Y)-1)≡ Y

In any of these operations, if Y does not occur in X, then the scan is said to fail, and the value of the resultant string temporary indicates this (see items 5 and 6 below).

4) The relative precedence of these new operators is:

:
;
arithmetic
concatenation
UPTO FROM AFTER BEFORE IN
relationals
logical connectives
assignment
:
:

5) The notation in 2 and 3 above can be used on the left-hand side of assignment statements in what might be called pseudoexpressions, which can be considered in the same class as PL/I pseudovariables. Expressions used in this way are not substantially different from using ordinary subscription on the left-hand side. For example:

```
X = 'THIS, I BELIEVE, IS TRUE';
X AFTER ',' UPTO ',' = '';
```

results in X containing

'THIS, IS TRUE'

6) Replacement is conditioned upon lack of failure in all expressions in a statement. For example:

```
Z = 'XYZ'; 'A' IN Z = 'MN';
```

results in Z containing 'XYZ', and the last statement is said to fail.

Assignment to an unnameable CS such as A||B = C; is also said to fail and will not be executed.

7) Replacement of pseudoexpressions in varying strings causes the designated string to be adjusted (expanded or contracted) to contain the expression on the right-hand side. For example:

```
X = 'ABCDE'; X UPTO 'B' = 'XYZ';
```

results in X containing 'XYZCDE', and

```
Q = 'PUT OUT THE CAT'; ' ' IN Q = '';
```

results in Q containing 'PUTOUT THE CAT'. It is also reasonable to adopt the same convention for fixed length strings. In this case the necessary adjustment to the target string would require extension on the right with blanks or truncation of those characters on the right which would not fit, in order to preserve the declared length of the string. The traditional interpretation is easily preserved by writing

```
X(:I...J) = Y(:1...J-I+1);
```

8) Two additional built-in generic functions should be provided: SUC and FAIL. When used without arguments they return a truth value (bit string of length 1) indicating the success or failure of the most recent attempt at string formation, and the indicator is reset to success. When used with a string expression as an argument, the corresponding truth value is returned. Examples:

```
X = 'PUT OUT THE CAT';

DO WHILE(SUC); ' ' IN X = ''; END;
```

results in X containing 'PUTOUTTHECAT', and

```
DIGIT = '1234567890'; ST = 'AB123D';

DO I = 1 BY 1 WHILE (FAIL(ST(:I) IN DIGIT)); END;
```

results in I containing 3.

9) There should also be a pseudovariable SUC which could be used to set the success/fail value.

10) The occurrence of an array as the criterion in a scan results in a left-to-right scan which is terminated by the first successful match of any element in the array. The longest array element is selected if two or more satisfy that criterion. (The inclusion of array constants in PL/I would greatly enhance this function.) Example:

```
DCL DIGIT (10) CHAR (1) INITIAL ('0','1','2','3','4','5','6','7','8','9');

X = 'ABC123DEF'; A = X BEFORE DIGIT;
```

results in A containing 'ABC'.

11) Success and failure are local to the block in which string formation takes place, although they can be passed between procedures as attributes of string arguments and results.

12) All function calls in an expression are evaluated, whether or not any previously computed subexpression in the statement has failed.

Discussion of the Proposal

1) It is possible that the notation shown for the five new operators is ambiguous, at least in a pragmatic sense if not a syntactic sense. This arises since they could be confused with identifiers, and PL/I is premised on a philosophy of no reserved words. One viable alternative would be to use an escape character in the prefix position providing that this notation is totally unambiguous. One such possibility would be

```
"UPTO "AFTER etc.
```

This would also fit nicely into a scheme for abbreviation such as

```
"U  "A etc.
```

2) It has been pointed out that the generality implied by the extensive use of left-hand-side expressions (pseudovariables) leads to a definite ambiguity with respect to the equal sign (=). It is worthwhile to note that this highlights the unfortunate situations which arise when a symbol has two distinctly different meanings, supposedly always delineated by context. Consider the example:

```
DCL A BIT(10),B BIT (1); A BEFORE B = '1'B = '1B';
```

An interim solution, which is far from adequate, would be to define the first occurrence of the equal sign in an assignment statement to mean replacement, and all others to be the relational operator, as is currently the case in PL/I. It is also possible to consider the same rule with the additional proviso that an equal sign in a parenthesized expression can imply only the comparison operation.

3) The meaning of replacement changes depending on the success or failure of all string operations in a statement. The definition included in the proposal says that failure implies no replacement. An alternative would be to have failure imply replacement by the null string, although exactly what should be replaced is not clear in every case. SNOBOL uses the definition in the proposal and its success leads one to support adopting that rule. Another alternative is to allow the programmer to change the mode of replacement by calling a built-in function.

4) It is interesting to consider the addition of an indirectness operator. This is being explored. One might imagine the following sequence of statements using the indirectness operator IND:

```
X = 'ABC';

Y = 'X';

Z = IND Y;
```

This results in Z containing 'ABC'.

5) A similarity can be noted between the operators introduced in this proposal and the traditional relational operators.

> AFTER
> = FROM
< BEFORE
< = UPTO
= IN
(¬= NOTIN)

It is important to note that the string operators return general string values, whereas the relationals return only bit strings of length 1 (logical values). The value of NOTIN is, if anything, always the empty string; the one situation in which it might appear useful is resolved by using ¬ in conjunction with SUC. For example:

IF ¬ SUC(X IN Y) THEN...

6) This proposal points out a few PL/I constructions which should be generalized. Already noted are the lack of array constants in other than the INITIAL attribute of the DECLARE statements and the use of one symbol to mean both assignment and equivalence. In addition, the ability to subscript arrays and string expressions, not just identifiers, would be valuable. One might consider the syntax

(array or string expression) (subscript expression)

which appears to be syntactically unambiguous. Alternatively, this function could be supported through the use of an explicit subscription operator for expressions, while allowing the traditional implied operator for identifiers.

7) Item 12 of the proposal is open to question, but it is felt that the solution offered is appropriate for most uses of CS facilities. Item 11 alludes to the passing as arguments of CS expressions which have failed. Their meaning in the called procedure would be identical to that of any other expression which has failed. Returning a value such as ' ' IN '' would pass failure back to the calling program.

8) With regard to implementation, it should be noted that CS temporaries generally need not be constructed, except as the result of concatenation and subscription generating reversal of the substring being specified. In all other cases, CS temporaries take the form of reference blocks (or "dope vectors") flagged as temporary which contain: a pointer to the string ultimately referenced, and the extents of the temporary as a substring of that CS.

Current Status and Conclusions

The proposal as outlined has been implemented as a set of defined operators and a dynamically allocated CS data type in the MAD language using its self-extension facilities (Reference 6). This package is available at Yale, the University of Michigan, and is being adapted for an earlier version of MAD under CTSS at MIT. It is expected that this effort will result in valuable response from users at these institutions across a wide variety of applications.

The proposals outlined here appear to be a consistent set of extensions to a general purpose procedural language. They do not constitute any significant functional extensions over what PL/I now provides. Attempts at implementation in MAD indicate that no significant problems arise; and that, indeed, the generality of CS expressions as opposed to the PL/I constrained notation can often lead to gains in efficiency (Reference 6).

Acknowledgements

The author wishes to thank the following colleagues not resident at Yale for their comments on draft versions of the proposals:

M. D. McIlroy
R. E. Griswold
C. H. Weisert
Mrs. S. Greene

In addition the aid and assistance of the computing center staff at the University of Michigan in installing these facilities in their system is gratefully acknowledged.

References

1) Farber, D.J., Griswold, R.E., and Polonsky, I.P., "The SNOBOL3 Programming Language," Bell Systems Technical Journal 45, 6 (July-August 1966), pp. 985-943.

2) Farber, D.J., Griswold, R.E., and Polonsky, I.P., "SNOBOL, A String Manipulation Language," Journal ACM 12, 1 (January 1964), pp. 21-30.

3) Yngve, V.H., "COMIT," Comm. ACM 6, 3 (March 1963), pp. 83-84.

4) Madnick, S.E., "String Processing Techniques," Comm. ACM 10, 7 (July 1967), pp. 420-423.

5) IBM Corp., "IBM System/360 Operating System: PL/I F-Level Reference Manual," Form No. C28-8201.

6) Rosin, R.F., "SNOMAD2," Memorandum 14, Computer Center, Yale University (August 1967).

Appendix

Four Brief Examples Using the Existing and Proposed CS Notation

/*1. IN STRING A WHICH CONTAINS AT LEAST 10 BLANKS, REPLACE THE FIRST 10 BLANKS WITH '*' */

```
/*  EXISTING NOTATION */

    DO I = 1 TO 10;
        SUBSTR(A, INDEX(A, ' '), 1) = '*';
    END;

/*  NEW NOTATION */

    DO I = 1 TO 10;
        ' ' IN A = '*';
    END;
```

/*2. REPLACE ALL OCCURRENCES OF '*' IN X BY '/' */

```
/*  EXISTING NOTATION */

    L1: I = INDEX (X, '*');
        IF I ¬= 0 THEN DO;
           SUBSTR (X, I, 1) = '/';
             GO TO L1;
        END;
```

```
        /*  NEW NOTATION */

            DO WHILE(SUC); '*' IN X = '/'; END;


/*3.  DELETING ALL BLANKS FROM STRING X */

        /*  EXISTING NOTATION */

            LOOP:  I = INDEX (X, ' '); IF I = 0 THEN GO TO DONE;
                X = SUBSTR (X, 1, I-1)|| SUBSTR(X,I+1);
                    GO TO LOOP; DONE:;

        /*  PROPOSED NOTATION */

            DO WHILE (SUC); ' ' IN X = '' ; END;


/*4.  THIS ROUTINE SCANS THE STRING T FOR THE FIRST SUBSTRING
      OCCURRING BETWEEN A PAIR OF BLANKS AND ASSIGNS THIS VALUE
      TO  X.  IF NO SUCH STRING EXISTS THEN CONTROL IS TRANSFERRED
      TO Y.  THE PORTION OF T UP TO AND INCLUDING X IS THEN
      DELETED.  */

        /*  EXISTING NOTATION */

            I1 = INDEX (T, ' ');
                IF I1 = 0 THEN GO TO Y;
              T1 = SUBSTR (T,I1+1);
            I2 = INDEX (T1, ' ');
                IF I2 = 0 THEN GO TO Y;
             X = SUBSTR (T1, 1, I2-1);
             T = SUBSTR (T, I2);

        /*  PROPOSED NOTATION */

            X = T AFTER ' ' BEFORE ' ';
            IF FAIL THEN GO TO Y;
            T UPTO  X = '';
```

Questions and Answers

Comment: (Mrs. Lasky) I'd like to make a comment. I know that you said this at the end of your comments, but I'd like to reiterate it. At APL we have written a program that translates FORTRAN to PL/I. Naturally this is completely string manipulation; all you do is scan a string, pick out different parts, and use the string manipulation facility of PL/I. We've also written a text-editing program and a print program, and are in the process of writing a program for the searching of records. This is all done using the string manipulation facility in PL/I. You may get tired writing SUBSTR, and you may want to have some notation that would make it easier and quicker to write, but you can do it—the functional ability is there.

Answer: I can give an example one generation earlier than yours. In a way that's not even so powerful as that provided in JOVIAL, we have the facility in MAD for manipulation characters—but really as things superimposed on top of integers. We have students who write little pieces of compilers in this. The second problem in my introductory course this year was conversion of messages written in Morse Code to English, and in English to Morse Code. It can be done, and it's relatively straightforward. But it would be much more straightforward in the facility I just proposed.

Question: Have you looked into the question of how big a problem it would be to use PL/I compile-time facilities?

A: The PL/I compile-time facilities which exist now allow extension of the language only in terms of functions, not in terms of operators. And I am the kind of guy who likes infix operators and not functions nested 15 deep to do things that are otherwise not straightforward. For example:

My proposal: `X = Y AFTER ',' UPTO ',';`

Existing PL/I: `X = UPTO (AFTER (Y,',',), ',');`

One thing one cannot do in PL/I is to define pseudofunctions or pseudovariables. But the compile-time facilities of PL/I, as it stands today, do not allow one, for example, to define operators. If it did I would have no quarrel, and indeed I would never consider proposing this as an extension to PL/I until I had used it

for two years. I think this is a great plug for real compile-time facilities, real macro facilities. I say that with the full understanding that no one, I think, is really sure exactly how to provide everything that everybody talks about in terms of a real macro facility. I think a lot of people come close, but then seem to leave a little off. Galler and Perlis have an interesting article in the Communications of the ACM that goes only so far, and it wouldn't encompass this proposal. Chetham gave a very good paper at the 1966 Fall Joint Computer Conference which is really excellent, and I think he gets down to the nub of a lot of this. But I think there are still a few things that seem to escape him — for example, the definition of additional data types, which turns out to be a knotty problem. The central problem in building extensible compilers in general — that is, code generation and the context-sensitive features of language. But I agree with your point, Chris. I built this proposal as extensions into MAD, and I could do this because MAD allows me to define infix operators and to define new data types. I can use these operators on the left-hand side of assignment statements.

Q: You've indicated that you spilled a little blood over this. Are you actually advancing this proposal or passing it on to . . . ?

A: This proposal hasn't been developed overnight. Most of you are seeing it for the first time, and it's rather muddy and unclear. It really takes several hours of thinking just to understand what the objectives are, and what one does with strings in a high-level language anyway, if you've never done it before. I have proposed this; the SHARE project has considered it; I have mulled it over with students and colleagues at Yale for over a year now, and we've worked through lots of examples; and we understand the problems of implementation. The SHARE project adopted a resolution which asked IBM to consider this proposal and either give us a date for implementation or consider and present alternatives to us. I can report to you that their response to this resolution was, "Not at the present time."

Again, I think this is a good case for an extensible compiler, but we have proposed it for the language. I think it is unfortunate that one would feel compelled to do this, that the language has to be extended in this way. Although Fred (Professor Brooks) isn't here to agree or disagree, I think if PL/I had had a few more

years to mature before it was thrust upon the world, it might have included a facility like this. I suspect that people like Griswold and Polonski and Dave Farber (who is now at RAND) —who did the original SNOBOL—would have contrived something like this. They were interested in PL/I; they were interested in writing a SNOBOL processor using the PL/I language—but they were, in a sense, never given the opportunity.

EXPERIENCE WITH PL/I

Earl O. Althoff
Director, Joint Computer Evaluation Group
Eastman Kodak Company
Rochester, New York

I am very happy to be invited to discuss PL/I with you. To make this discussion more meaningful, it is important that you understand my point of view. Every time I think of this matter of points of view, it reminds me of the story about the two drunks who were trying to walk home from the local tavern, but, in actual fact, made a wrong turn and ended up crawling down the railroad tracks on their hands and knees. One drunk finally said to the other one: "Say, did you ever climb so darn many stairs in your life?" The other drunk answered: "No, but it wouldn't be so bad if it weren't for these low banisters."

Well, this may be an unusual view of a railroad track, but it does illustrate the point I am trying to make. That is, reports on PL/I seem to differ, not from intentional bias, but often because the reporter doesn't have a really overall view. Getting back to PL/I, I think I have been in a very fortunate situation, since I have really been concerned with it from several views over the last four years. That is, professional activity as a GUIDE officer, an overall company view from the standpoint of a person working at the company level with worldwide standardization responsibilities, the points of view of many individual company units representing practically every kind of previous language usage possible, the point of view of the PL/I programmer, and the training view of those just starting into PL/I.

Professional Activity as a GUIDE Officer

GUIDE, the organization of large-scale commercial IBM users, has a PL/I project of about 70 members who are working with IBM on PL/I development. This effort emphasizes commercial usage and is complemented by a SHARE project. The two projects meet jointly at least four times per year.

I have been active as an officer in GUIDE and am currently president. A report I made to the GUIDE membership on May 18, 1965, seems as true now as it did then. I'll read the first part:

> "Otis Simpson [president of GUIDE at the time] has asked me to review with you at the beginning of this meeting the activities of your GUIDE executive board pertaining to PL/I. We might begin by asking why we, serving you as elected representatives of all GUIDE members, are closely watching these trends in language development.

EXPERIENCE WITH PL/I

"I can assure you that I am not here to praise PL/I, or to disparage any existing languages; I am here to report simply on trends. One major reason that is advanced for PL/I is that it is inevitable that, some day, a computer language must be developed which can do all of the jobs of the many partial languages in use today. These partial languages such as FORTRAN, COBOL, SPS, Autocoder, RPG, MAP, JOVIAL, LISP, and the like have made a valuable contribution. They are today used for our live programming and have contributed also to our total knowledge about languages. However, many students feel that if one attempted to expand any of these partial languages into a total language, he would end up with such a mongrel result as to invite almost certain failure. The second major reason advanced for PL/I is, of course, that it can be changed quickly to include new features. This is particularly important in our rapidly advancing technology.

"I am sure we would all like to have an absolute written guarantee that PL/I will be this pluperfect total language that is vitally needed today. Lacking this guarantee, your GUIDE executive board has taken a careful and deliberate approach in determining GUIDE's best role in relation to PL/I. Initially, it was uncertain as to whether the language was an experiment, or whether it was indeed a serious effort. With further developments, it has now been determined that 1) PL/I is a very serious effort indeed, 2) the fundamental objectives of PL/I are to produce a total language and to incorporate new developments at once as described earlier, 3) IBM has promised that they will fully incorporate the commercial needs as determined by GUIDE, and 4) PL/I is <u>the</u> effort for the third generation of computers, and it is where the language development money is <u>going to be spent</u>. Thus, we see that if GUIDE members are willing to volunteer now to contribute the benefits of their experience for PL/I, we can have simultaneously with the total hardware concept of the 360, a language that will be as total as our current knowledge permits.

"As is usually true in these cases, one is never quite certain whether or not the time in history brings about the development, or whether a given development brings about history. However, now is a historically correct time for users to consider a total language alternative. In the first place, improved technology has forced upon us the adoption of a new computer command structure and a radically different data structure. We as users may have

not desired this change, but today's needs and the price/performance ratio are such that many of us are undoubtedly going to make a changeover. Since we have a reprogramming task, or more hopefully a translation task, whichever language we move to will involve about the same amount of effort. In fact, the total language, PL/I, should be much easier to mechanically translate to than any of the partial languages. As a caution, however, it does appear that PL/I will lag 6-9 months behind the earliest available 360 language choices.

"The second reason for considering going to PL/I now is that we are entering a new era where commercial EDP will be vastly expanded in scope and complexity over our earlier concept of purely business EDP. We need a language that integrates engineering and business needs, a language that is good for real-time applications, a language that can be added to and modified very rapidly, a language that is good for teleprocessing and man/machine interaction, as well as a language that incorporates all of the basic capabilities of assembly language, Autocoder, and COBOL."

GUIDE started PL/I project activity with a comprehensive analysis of what was needed, and I helped author a report dated April 27, 1965. I recently reviewed this report so as to judge progress of PL/I versus our report.

Several of the items in the report pertained to the need for strong user activity in the areas of understanding, usage, experience, and training. The cycle here is that early users make the language work in their own shop; then they exchange experiences to develop a consensus; and last, but not least, IBM publishes user-oriented manuals and training courses.

Consider terminology: In 1965 all we had was the study group report—full of technical and hard-to-understand terms. Good commercial names for PL/I features had not even been thought up yet. Today, several books and manuals explain PL/I quite understandably.

Consider training: A higher language removes the programmer that long mile away from the machine, so he loses identity with object code unless he can study good material with many examples. So we had many stories by early PL/I users of jobs taking 5 or 6 hours at first that were cut to 10-20 minutes by study of object code. This period is over now, since it's fairly easy to pick up coding tips and thus write effective programs from the beginning.

In summary, devoted effort on the part of early users, and their willingness to share ideas and experiences, have largely overcome problems pertaining to terminology, understanding, usage, and training. IBM has recently published a very good manual, and programmed instruction and other training material is beginning to appear.

Now let us turn our attention to language items. The main need cited in the 1965 GUIDE report was lack of record-type input/output. The language simply could not be used without it. IBM listened and added Record I/O on a high-priority basis. The result: PL/I users now enjoy Record I/O in addition to Edit-Directed I/O and Data-Directed I/O, and have Locate Mode from List Processing coming along. In only two years we have advanced from an unusable position to where one has the combined options of three single-purpose languages —certainly, input/output capability is one of PL/I's outstanding features.

The GUIDE report also stated that certain built-in functions, such as sorting, selecting, merging, and summarizing, are highly desirable. These would work just like "Sine and Cosine". As of today, only initial study has occurred. IBM recently promised that PL/I products would cover data base capability, so we can assume this will now be given a high priority for language invention.

All of our GUIDE projects have fairly lengthy priority lists of wanted features and PL/I is no exception. It seems like, as fast as several are taken care of, new ones appear in our rapidly advancing technology.

Time does not permit me to make a detailed description here, but I could say a few general words and perhaps there will be time in the discussion period if you have further questions.

First, we are very concerned with machine independence as a basic principle. The GUIDE and SHARE project groups, and to a major extent also IBM, are really zealous on this point.

Second, the GUIDE PL/I project reported last November that PL/I was, today, an ideal language for commercial data processing. SHARE backs it for scientific purposes, and, in fact, the project leader of the FORTRAN project switched over to become project leader of the PL/I project to help enhance conversion methods devlopment.

Third, the next release in June will essentially be a full-language compiler. So now the GUIDE and SHARE projects are working mainly on advanced features not in any languages today—data base needs, teleprocessing needs, decision tables, code translation, items helpful in writing software.

EXPERIENCE WITH PL/I

The 1965 GUIDE report also cited compiler features. The needs for cross-reference listings and some facility which equaled the COBOL copy facility were stressed. The people writing in PL/I report that the compiler, together with its compile-time facility, already goes far beyond the capability mentioned in the GUIDE report.

The need for the compiler to be integrated with a total testing system was also stressed. Here we mean a PL/I version of Auto-test. The language debugging features of PL/I are truly outstanding. However, lack of the "testing system" has meant that this need could not be satisfied in the manner inteneded in the GUIDE report, but we do now have a GUIDE group working in the area of a testing system.

Back two years ago, the greatest GUIDE effort was spent on getting across to IBM the need for fast compilation time so users could make satisfactory EDP progress. In particular, a cost analysis was given to IBM. It was stated in this cost analysis that at $4.00 and less per compilation, source language debugging was economical. From $4.00 to $8.00, costs were such that compilations were not restricted, but simple corrections were patched. Between $8.00 and $15.00, it was found that the compiler could be used, but EDP management restricted usage and demanded grinding hours of desk checking — so EDP progress was held back.

Another point is that until costs are below $10 per compile, almost the sole selection criterion of a language is compile time, since differences of $10-20 per compile dwarf language facility savings. Below $10.00 and particularly below $5.00, the major language selection criteria are language features and capability, since a difference of $0.10-$0.20 per compilation is meaningless if programmer time is saved.

In summary, the GUIDE report stated, in 1965, that compilation costs of around $5.00 were essential if rapid EDP progress was to occur in our companies.

It is a little difficult to give a simple conclusion as to where we stand in relation to this goal. For one thing, the cost per compilation of the F-level compiler depends on 1) the amount of memory allowed for the compiler partition, 2) main frame speed, 3) the hardware peripherals, and 4) the number of features you elect to use. We compiled a program of 1,000 statements on a Model 65 with HASP, in 159K, and using full cost rates, we were able to compile from a cost of $3.00 up to $9.50 depending on how many features we elected to use.

Certainly, we are at the range where source level debugging is economic provided that programs are 500 statements and less. There are two ways of

looking at this. You can establish standards of 300-500 statements per PL/I module so as to be in the economical range for source language debugging, and then link modules together. The second way to look at it is to assume that we will get faster compilers, and then use standards of 800 to 1,000 statements as your limit.

GUIDE, in this early study, also stressed object program efficiency—particularly pointing out that about 70% of the time our commercial programs are doing a very few simple things, such as moving data, an equality comparison, an unconditional GO TO, a subroutine linkage, and simple addition. Hence, if anyone designs a language and compiler so that these few items can be as efficient as with Assembler, one could write programs of 80% efficiency and up, compared to Assembler.

The early versions were rushed out to help committed customers meet urgent target dates. IBM then concentrated on efficiency for Release 3, which came out last November.

With Release 3, if one could develop a totally new system, and if an experienced person designs his record structures properly so that data conversions are practically nil, and if we restrict our coding to a suitable subset of PL/I, we are indeed up in the 80 to 90% efficiency range. However, the cost to write this type of program is too high for general usage, so we feel that a greater percent of PL/I must be handled efficiently over the long haul.

I could sum up this professional veiw by paraphrasing a proverb learned at Kodak Germany: When I look back at 1965, I am amazed at the length of the road we have come; but, when I look at the length of the road ahead, the road behind looks quite short in relation to the road ahead. The road ahead includes many advanced language inventions, since the nature of EDP is that of a dynamic environment and language must keep pace with hardware. The road ahead also includes conversion tools and industry standardization, each of which are major efforts.

Overall Company View

Now turning to the company view: We decided in late 1965, on the basis of a thorough study and IBM's commitments, to go to PL/I. We stated in 1965 that we expected to do initial study in 1966; we would do pilot work in 1967; and in 1968 we expected to have a good production tool. This timetable was chosen after looking at the entire complex of hardware problems and expected delivery

dates of OS and PL/I. In fact, of the three, PL/I has been in far the best shape compared to our needs.

Before you take "pilot work" to mean only an occasional program or two, let me explain that this is not what it means. By "pilot" we meant that target dates could not be guaranteed in the sense that our 7080 target dates could be guaranteed. But, if crucial target dates did not exist, we would write in PL/I. In addition to this, one installation used PL/I on an absolutely rigid target date on a very large integrated commercial system. This ensured that solid company experience was gained during 1967. In fact, the latest Rochester statistics I have show that we have 21 jobs on a Model 65 running nearly around the clock on PL/I-oriented commercial work. The computer installation which concentrates almost entirely on scientific-type work reports that around 40% of their work is now done in PL/I.

The judge of a language is how closely the selection criteria are being met. Hence, I will discuss PL/I selection criteria, and then cite Kodak experiences in relation to each.

First is the ability to do all types of one's work. Here, PL/I has certainly been a resounding success. Some 21 different commercial applications are either in production or in the final stages of testing in Rochester alone. Examples are a large job status information system, an inventory system, and a capital asset system. In the area of scientific computing, almost all of our scientific groups have established PL/I expertise. And it is thus being used on a wide range of jobs, mathematical, statistical, simulation, quality control, standard time and cost, etc.

Of particular interest, however, and getting more directly to the reasons we feel that PL/I is the only language that truly complements 360 concepts, is what we have learned from combined scientific-commercial efforts. In one system, we are entering case weight data directly into the computer. The scientific people are, of course, experts in work connected with scales, packaging, and operating efficiency; meanwhile, the commercial people are getting the weight information into the necessary shipping and billing documentation. Both groups are able to code in PL/I in an integrated manner.

In another major plant, a large-scale information system was written principally by the industrial engineers (from the so-called scientific community); yet this was integrated with commercial systems through the help of commerical PL/I programmers.

EXPERIENCE WITH PL/I

In one installation, a manpower emergency in a group preparing a commercial application was met by a temporary transfer from the Research Laboratory. Under prior concepts, this person could not have been effective for months, rather than immediately, so target date setbacks would have been the only possibility.

The people on these and other similar projects report that use of PL/I really saves by giving the ability to truly communicate with each other.

A second selection criterion is the ability to use a language at all of your locations. One of the factors involved here is acceptance. We have had no difficulty with acceptance of PL/I as a language. Several units are using COBOL, at least temporarily, because the PL/I compilers were not available in time. The 1130 people are using FORTRAN only because IBM has not yet offered a compiler for the 1130 in PL/I syntax. All in all, this selection criterion is being satisfied.

The third item concerns influence on the language. Here, if a language is under your absolute control, you would score a 100% on this point. With no influence, you would score zero. By working on the PL/I project, and the SHARE project, and also even as an individual customer, we have indeed been listened to by IBM. I can't say that every item that we have asked for has been immediately accepted, nor can we expect it to be. But we would certainly have to say that PL/I scores a tremendous plus in regard to the selection criterion of influence on the language.

Another selection criterion is extent of integration with other software. Here 100% would mean, for example, that every feature of OS is immediately available easily through PL/I. A score of zero means that PL/I is considered as a language entity far apart from OS features. Here we consider PL/I to be improving and certainly better than other languages, but short of requirements. The main problems are in the areas of multiprogramming and teleprocessing.

The next criterion concerns extent of usage throughout the industry. Here a language rates 100% if everyone uses it, and a rating is zero if you are the only person using it. PL/I was, until Version 3, a language still under early development, so the percentage of users is in line with our expectations. Other manufacturers are committed to PL/I compilers, while others are in various stages of PL/I development, and there is quite a community of users.

Of critical importance to the community-of-users criterion is that the users should come from all types of computing; in this regard, as far as we can tell, there are relatively equal numbers of scientific and commercial users.

Of course efficiency is "the" criterion. Efficiency is a complex problem and refers to a number of factors — compiling speed, object program speed over an 8-10 year period, core and memory efficiency, source level debugging capability, testing time, turnover of programs, ability to use a scientific person on a commercial job, modular programming capability, turn-around time for the compile and test cycle, and a lot of more mundane things such as even keypunching time.

Since it is difficult to get at an evaluation of efficiency over this range of factors, let us start by looking at the view of one of our units as to total efficiency. At the current time, this group has 60 man-years of systems design and programming experience with PL/I. A major production application, invoicing and sales accounting, went in full production on a live basis November 1st. The project has 70 PL/I programs, approximately one-half of which are used for auxiliary purposes. The F-level compiler was used on 128K equipment.

They report that, setting a modular standard of 350 to 400 statements, the compiler gives a competitive cost/performance when approximately 96K is used for the partition size.

With manual optimization of some programs, and with standards which avoid some PL/I features, and with Version 3 of PL/I, they report that their total costs are below the costs of the previous computer systems.

This is on a 360 that has been deliberately kept as small as possible in line with the load, so it has minimal memory, no multiprogramming, and no 2314 equipment. Here we have a major effort now in full production which has lower running cost than the prior system, with what appears to be a certainty of further cost reductions.

The usage of PL/I has permitted work to go continuously forward in spite of turnover. In their 12 years of computing experience and on this extremely large job, the smoothness with which the individual programs were put together in one run, were then tested in parallel, and then were launched into production, has been an order of magnitude better than any other effort. PL/I must take a great deal of the credit.

I would like to compare costs for you with other language choices, but I have three problems: 1) we haven't duplicated any 20-30 man-year effort, which is the only true test; and 2) we would have to compare a single language versus two languages plus more Assembler, which would be unfair; and

3) we would have to know for certain the extent of compiler improvement over the 10-20 year life of the system.

I listened to experiences cited by a number of people; PL/I is today either lower in cost, about the same, or higher—you can take your pick. Since this approach didn't work I went back and assumed that IBM would, as officially stated, offer new PL/I products. I had to assume something, so I merely assumed that any speed and language facility now in other compilers or talked about for PL/I could be expected over time.

It was easy to conclude that, with all of these goodies, we could then save around 10-20% on work that could be done in a single-purpose language, 20-40% on mixed work, and an additional major savings if the software capability problem is solved.

For D level, where we are at liberty to redesign, we programmed one job in 1.6 times as much memory as with 1400 series, or just a little greater than is required for Assembler. Where we cannot redesign from stem to stern we are finding that three to four times as much memory is required. Experience is still too limited for overall cost analysis.

To summarize the total company view, good strides have been made towards achieving the objectives of PL/I selection. Economical work is being turned out, except in the portion of scientific work having high compiles. We are looking forward to greater EDP progress at lower cost as new PL/I products become available.

Experiences of Company Units

My third set of views comes from working with each unit as they individually evaluate hardware and language choices. Each unit is autonomous and makes its own choice as to when and whether to use PL/I. Many units like to use the type of chart shown on the next page.

Probably most of you are familiar with this chart selection technique, which often reduces a judgmental and seemingly controversial problem covering a wide range of factors to a choice upon which all will agree. That is, for each item concerning a language, we have found that people will argue endlessly as to the pros and cons and further possibilities. However, since all we are really concerned with is the actual final choice, a weighted table may show that all agree on the final choice, even though everyone disagrees as to the relative reasons why. Usually a representative sample of five people or so give their ratings and these are then averaged.

EXPERIENCE WITH PL/I

I don't suppose that any of us here would write down the same specific weights and ratings that the author of this particular sample did. However, the important thing is the overall conclusion. Here we have found that it has not made any difference what weights the various units have given to each factor, or what people did the rating; the conclusion is always clearly in favor of PL/I.

SAMPLE LANGUAGE SELECTION TABLE

	Weight	Assembler and Macros	COBOL	PL/I
A. Machine time factors (over next 5 years)	(40)			
1. Compilation time with program master	15	8	12	14
2. Object program (3rd generation features)	15	15	5	10
3. Testing efficiency	10	2	6	10
B. Programmer factors (over next 5 years)	(32)			
1. Transfer of programs	12	2	12	11
2. Coding-debugging time	14	3	9	12
3. Learning time	3	2	3	3
4. Morale-acceptance	3	1	2	3
C. Machine independence	(13)			
1. Convertibility cost	8	0	5	7
2. Extent of compilers offered	5	0	5	3
D. Other factors	(15)			
1. English-documentation	4	1	4	3
2. Scientific and comm'l use	3	0	0	3
3. Company compatibility	5	2	2	5
4. Speed of incorporating new concepts	3	3	1	2
Total		39	66	86

So, each unit has rated PL/I as the best language. But there is a question of when? Perhaps some of you have heard one consultant who makes the jibe

repeatedly: "PL/WHEN". Our units would say, generally, that this should be "PL/NOW". There is no real problem in a new case.

However, if one has a small staff who already must support second-generation Autocoder and Assembler languages, COBOL, FORTRAN, and 360 Assembler, one has trouble. We have found it to be too great a burden for a person to keep viable on over two major languages. In this type of unfortunate case, we must first phase out a couple of the older languages, so some of our units may not be able to start PL/I usage until late this year or early next year.

Programming with PL/I

The fourth view I've been involved with is the programmer view. Having spent a good many years in both scientific and commercial programming, this point of view is dear to my heart.

And I suspect most of us, no matter what we might put on these language selection charts concerning other factors, have the question of whether or not programmers will be enthusiastic about the language uppermost in our minds as we are deciding which language or languages to employ.

Having talked to well over 100 programmers in the company who are using PL/I, I can assure you they are most enthusiastic about PL/I. Our former machine assembly language programmers have taken a couple of months to build up this enthusiasm, since a higher language is a little offensive to their sense of striving for the utmost in machine cycle utilization. However, it has generally been almost immediate on the part of programmers who have previously used higher languages.

It is much more difficult to pinpoint just why these programmers are so enthusiastic in order to explain it at the management level. It often appears that a programmer either likes something or he doesn't like it. But I will indicate the most common reasons that programmers have advanced. Commerical programmers are very much concerned with debugging and with meeting target dates. The debugging features in PL/I are, as mentioned earlier, outstanding. Secondly, there is a noticeable uplift in overall morale when a group shifts to PL/I. Part of this is due, I think, to the fact that other college graduates, engineers, and scientists, including PhD's, are visibly using the same language. We are also starting to get accountants and other commercial people to do some programming in PL/I. This tends to remove the second-class stigma where our systems analysts have felt that

they are looked down upon as mere technicians since they are programming. Another factor is the ability to change jobs between so-called scientific and commercial projects.

The next item is true for almost any higher level language in the third generation; one simply cannot patch. The hatred of patching occurs primarily from the experience of taking over a program full of patches or hunting for an error when another programmer is on vacation—it's no fun. The programmers also like PL/I simply because it is new and exploits new concepts. I also find that programmers are very concerned about the high degree of clerical-type work in their job. Certainly PL/I reduces this another level by use of the default options.

Again, there are many programmers who like to feel that they are working with the best tools, the best language, the biggest computer, etc. Certainly the list of additional capabilities and features in PL/I over prior languages is imposing, even apart from the concept of a general-purpose language.

Time does not permit me to go into the superior language features in any great detail and, for that matter, I do not regard myself as a language comparison expert. However, just let me give one example. In one of our prior commercial languages, a commercial programmer who was not a mathematician was faced with the task of taking a square root in order to calculate an inventory formula. It actually took him approximately 2 months to study square root and get it programmed and debugged in his commercial language. He naturally did not regard this as challenging work or anything of that sort—simply as work which must be done and yet somehow senseless since he knew it had already been programmed by the scientific people in other languages. Needless to say, this programmer is quite happy to be working in PL/I, where we have square root as a built-in function.

Training and Starting to Use PL/I

Now turning to the last view, where I have been involved with many units helping devise plans to get started in PL/I. What should installations do who will be just starting to use PL/I now? First and foremost, our experience is that you can gain nearly as much out of rethinking your total systems and programming environment as you can from the greater capability of the PL/I language. Rethink the way you document, the way you test programs, and the way you run programs in production.

Thus we recommend giving your experts a few months' head start to develop new standards right from the beginning, incorporating into these all of your previous experience.

The next point is to have your IBM system engineer round up (from various users and internal IBM circles) the latest user-oriented papers on writing in PL/I, in addition to IBM manuals.

Next, start your first PL/I programming with a definite subset and objective in mind. That is, do not let your programmers assume they are trying out, experimenting with, or evaluating the language. The language permits so many ways of doing things that you may lose several months by playing around with the language unless production goals are set.

Let me say that I do not have the answers on how best to train people. In one commercial case, we simply taught elements of PL/I in one day and they started writing under careful guidance, gradually building on the one-day subset. That is, PL/I really is based on a simple language approach—one can teach six statements and be writing fairly complex programs that will run. After gaining confidence, other statements can be added. I recommend this approach for 1400 series size installations.

In other cases, we use three weeks of PL/I training followed by three weeks of practice exercises, thus turning out a fully functioning PL/I programmer. This is used for large-scale installations, mostly college graduates. We have also tried many shades in between. All of these methods work, provided the training methods are consistent with the target dates and purposes of your first selected jobs.

Lastly, I would advise having a specific individual appointed as a PL/I "expert". After some experience, he will be able to look over the proposed plan for a program, and quickly identify poor practices. He can also help where the PL/I rules may not agree with what a programmer would write on the basis of his prior experience.

I would like to close my remarks on this point of view of how to get started in PL/I with a word of warning. One should make organizational changes to fit PL/I. The programmers knock these PL/I programs out fast; hence the percentage of programmer-coders to systems analysts is sharply lower. Also, groups outside the Computer Department can and do use the same language. Only FORTRAN has been successful in this area prior to PL/I. This can be a trap in one way, but can be a real boon if you plan for it properly.

EXPERIENCE WITH PL/I

The real user outside the department can talk to computer people at the program level—a really significant advance on commercial work.

Questions and Answers

Question: Would you comment on the minimimum amount of core size in words or bytes for PL/I to be a viable tool in a commercial study?

Answer: For large-scale commercial work, we went live using about 100K out of a 128K byte machine. This pinches you a little too tightly for large-scale jobs; therefore, I would say 130K. Now for the small-scale jobs—say taking the typical 1401-type of problem on a 16K 1401—we are running these on 32K equipment in some cases, but I would say that a 48K Model 25 would be well chosen.

PL/I AS A TOOL FOR SYSTEM PROGRAMMING

Fernando J. Corbató
Professor of Electrical Engineering
Massachusetts Institute of Technology
Cambridge, Massachusetts

First I have to give some background, because I don't think you can discuss or evaluate a language like PL/I unless you know the background of the speaker. To some extent, PL/I is like getting too close to an elephant. All you can see is the pores, and what you see depends on which side of the elephant you're on. For present purposes, I have the advantage of not being a language expert. Instead, my vantage point is that of a system designer-implementer concerned with the overall system performance and the degree that the system meets the goals it was designed for. This gives me a little more detachment from the issue of whether the language is just right or not. For that reason some of my remarks will not be completely unequivocal but, rather, will be shaded. The basis of the PL/I experience that I wish to talk about is mostly on the Multics project, which is a cooperative project being done with the Bell Laboratories, the General Electric Company, and Project MAC of M.I.T. using the GE 645 computer, which is derived from the GE 635. However, I am not giving an official Multics view, but only my own opinion as a member of the design team. In fact, it's a preliminary view because things are too confused at this point to really be certain that we have analyzed what is happening. (A bit like asking for comment on a battle while it is still in progress; it's too early to know all the answers.) Further, one has to be cautious in forming final judgments on a language even though it is already a de facto standard, since there still is a need for a great deal of diversity in the computing field so that different techniques can be evaluated.

So that you can understand the context in which our system programming was done, I first have to give you a brief review of what the Multics project's goals are. A set of papers are in the 1965 Fall Joint Computer Conference Proceedings if you wish more detail. Briefly, we are trying to create a computer service utility. In particular, we want continuous operation of pools of identical units. We want to combine in a single complex the goals of interactive time sharing and noninteractive batch processing. We want to combine the goals of remote and local use in one system. The system programming problem is to develop a framework which multiplexes all this equipment at once and yet allows controlled interaction and sharing between users working in concert on various problems in real time. In short, it's a fairly ambitious project, not because of any single idea but because we're trying to tie together all these ideas at once. It was our judgment that we required new hardware to meet these goals squarely, and that, of course, meant that we had to write almost all of our software for

ourselves, including, it turned out, even the assembler. We were able to borrow a little, but not as much as we had hoped. Thus the project began basically at a research and development level, where flexibility is needed. In our view we wanted a small team of people, because the hardest thing to do when you're groping in an unknown area is to coordinate people. We felt strongly that we had to have maximum flexibility in our implementation.

To give you a little bit of the scale of the project, I will discuss briefly the implementation. The project began in earnest in the fall of 1964 and should develop a usable pilot system about the end of this year. This means it will take approximately four years to create a useful system. That's a long time, and I think one has to appreciate the investment of effort that goes into such a venture. If you spend four years developing something, you probably try to exploit it for a period of time greater than that; thus there is clearly an underlying goal here of wanting to see the project evolve as conditions change. The system itself is described quite tersely at the level suitable for a senior system programmer in about 4,000 single-spaced typewritten pages in the Multics System-Programmers' Manual. The system in final form seems to project out to about 800 to 1,000 modules of maybe four pages of source code each on the average, or, in other words, between 3,000 and 4,000 pages of source code. (It's interesting that the amount of description approximates the amount of code, but they're of course written at different levels.) The amount of system program that's in machine code is less than 10% at the source code level; it would be even less except that the compiler did not come along early enough, so some things had to be written in machine code right away. The system projects out to between one and one and a half million 36-bit words, which loosely is the supervisor program. In operation most of it, of course, pages out and is not resident in core, but it is expected to be there, and it is exclusive of all the languages and facilities of that sort, such as COBOL, FORTRAN, and even PL/I itself. The manpower to create the system has ranged from approximately zero to 50 people over a four-year period, with roughly an increasing number as time went on. When I say zero to 50 people, I mean effective persons who are involved and working full time. That isn't much for the size of the job that I've described, and there is clearly the need to have maximum leverage at the fingertips of each person.

The next question I want to address myself to is why one uses a compiler at all to do system programming. (I'll take up next the question of why PL/I in particular, but first: why a compiler?) First, there is the ability to describe programs briefly and lucidly. One can clearly obfuscate one's ideas with a compiler language, but it's harder. To some extent one is talking about what one wants rather than how one wants to do it. The trouble with machine code, of course, is that when one looks at a random section of machine code

one doesn't know what properties of the instructions the programmer really wanted to exploit. On the 7094, for example, the fact that the P-bit got cleared by an instruction may or may not be germane to what the program is trying to accomplish. With a compiler language, especially the later ones, one tends to describe what one wants to accomplish in terms of a goal and let the compiler work out the specific detail. This contributes to lucidity, of course. It also gives one the chance for change and redesign, because on a system as large as the one I have just described, the only sensible attitude is to assume that the system is never finished. Although the system obviously goes through phases, one is continually improving and evolving it. We have had this experience on CTSS, our previous time-sharing system, and we know it is true. What happens is that users keep having expanding needs and goals as they exploit the facilities, and they continually come up with wanted improvements. Certainly the other extreme—of assuming that a computer software system is like hardware and can be designed once and for all on a one-shot basis and then left to the hands of some maintainers—I think has been shown to be a failure.

Another issue, too, in a system of the ambition that we are talking about, is that the software is at least three-quarters of the design work and yet it usually doesn't get started until the hardware is already firm. Thus there is a desire to speed up the implementation effort, and using a compiler allows each programmer to do more per day. It's our experience that it doesn't matter too much if one is dealing with assembly language or compiler language; the number of debugged lines of source code per day is similar. Another point is that the supervisor is the host of the user services, so that the computer time spent in the supervisor is between 10% in some well worked out systems to maybe an extreme of 50% of the total time. Thus the possibility that the compiler isn't generating the most efficient code isn't a disaster. In other words, one is dealing with code that isn't being exercised all the time. It has to be there, it has to be right, but there is room for some clumsiness. Further, if the system is well designed, the production job will run efficiently and the supervisor will remain out of the picture.

Finally, there is the issue of technical management of programming projects: the problem of trying to maintain a system in the face of personnel turnover and in the face of varying standards of documentation. Personnel turnover is expected on a four-year project. (We didn't think it was a four-year project to begin with; we estimated two.) One has to assume in most organizations somewhere between 10% and 20% turnover per year even if everybody is relatively happy. People get married, husbands are transferred, and for a variety of personal reasons people must leave, carrying with them key knowhow. Training a new person involves a minimum period of six to nine months, even starting with good people, especially if you're faced with a system which has

4,000 pages of description in it. You don't casually sit down and read that, even in a weekend. In fact, it's fair to say that the system is large enough that no single person can remain abreast of all parts at once. Thus there is a reasonable case for a compiler in developing large systems.

In developing CTSS we used the MAD compiler slightly, and it was quite effective. The only problem was that we were cramped for core memory space for the supervisor. The compiler-generated object code was somewhat bulkier than hand code, and this was, unfortunately, a burden we couldn't carry too well; but where we used it, it was very effective.

So the question was: What compiler to use when developing Multics? We chose PL/I. The reasons go somewhat like this. One of the key reasons that we picked the language was the fact that the object code is modular, that is, one can compile each subsection of the final program separately, clean up the syntax, and test it on an individual basis. This seems obvious, perhaps because it's in several languages, like JOVIAL, FORTRAN, and MAD, but it isn't in some of the ALGOL implementations and it blocked us from considering the ALGOL implementation we had available. The second reason for picking PL/I was the richness of the constructs, especially the data structures and data types, which we considered to be very powerful and important features. We had an unknown task on our hands with fairly strong requirements. We viewed the richness as a mixed blessing, however, because we certainly were a little wary of the possible consequences. But it certainly seemed the right direction to start and maybe to err on and to cut back. As I'll get to later, it was a little too rich——but I'll come back to that. A third reason for choosing PL/I was that it was approximately machine independent. Our object in doing the system has not been to compete with normal manufacturing. Instead, our object has been to explore the frontier and see how to put together effectively a system that reaches and satisfies the goals that were set out. We are trying to find out the key design ideas and communicate these to others regardless of what system they are familiar with. Hence a language that gets above the specific details of the hardware is certainly desirable, and PL/I does a very effective job of that. In other words, it forces one to design, not to bit-twiddle. And this has turned out to be one of its strong points.

Another reason that we considered PL/I was that we thought the language would have wide support. To date it has had the support of one major manufacturer. And, the final key reason for PL/I was that two persons associated with the project, specifically Doug McIlroy and Robert Morris at Bell Labs, offered to make it work on a subset basis; they also offered to try to arrange for a follow-on contract with a vendor for a more polished version

of the compiler. That is basically why we chose PL/I. We certainly debated other choices somewhat casually, but these were the essential reasons why we picked the language.

The subset that was implemented initially as a quick-and-dirty job was called EPL, for Early PL/I. Its design characteristics went briefly as follows. It had no I/O; after all, this is a system programming language and we use the system subroutines. It had no macros except the INCLUDE macro, which worked in very smoothly with the time-sharing system, CTSS, that we were using. It had no PICTURE attributes or things of that sort which represented the COBOL influence, except for structures, of course. It had no multi-tasking; we found this to be a defective idea in the sense that it wasn't thought through well enough, and we certainly didn't need it for a system programming language. It had various minor restrictions like requiring structure names to be fully qualified. No complex arithmetic, no controlled storage (you can simulate that easily), and, more importantly, no attributes such as IRREDUCIBLE, REDUCIBLE, ABNORMAL, NORMAL, USES, or SETS—those things which allow the compiler to do an optimum job of compiling the code with advice from the program; these are sophisticated and tricky attributes, incidentally—but the reason they're not there is that the compiler didn't intend to optimize anyway, so it would have ignored advice.

To emphasize the positive, the things that EPL did have were ON conditions and signals; it did have recursive procedures—in fact, the system doesn't allow any other kind easily. (If you want to work at it, you can program a non-recursive procedure.) It did have basic storage and pointer variables, and it had ALLOCATE and FREE. It had structures, as I've mentioned, it had block structures, and it had varying strings, which we regret to some extent because of implementation difficulties. In other words, it was a pretty potent subset from the point of view of language facilities.

The implementation, as I said earlier, was deliberately a quick-and-dirty job. It was expected to be merely a temporary tool to be soon replaced by a polished compiler from the vendor. The team consisted of McIlroy and Morris and two to four helpers. I am going to give a detailed and candid account of the events surrounding the EPL implementation because the nature of the events, together with the very high qualificationsof the people involved, points out clearly that the difficulties encountered were quite unusual. The original optimistic estimate for making EPL work was that it was only going to take them about six months. In spite of the dedication of the people involved, it took them over 15 or 16 months to get a compiler that was barely usable. A lot of work has gone into upgrading it in the last 18 months, since the polished compiler of the vendor never materialized, and the upgrading process has not yet ended. Moreover, the EPL effort, like a grueling relay race, has worn out nearly

everyone who has worked on it. But to everyone's credit, the compiler works and is useful.

The language that was used to implement EPL was TMG, short for "transmogrifier", which is a language system developed elsewhere by Bob McClure. It's a clever, interpretive system specifically designed for experimental language writing or syntax analysis. However, it is not easy to learn and use and, therefore, it is hard to pick up the work of somebody else written in the language. The EPL translator was initially designed as two passes, the first one being principally a syntax analyzer and the second one basically a macro expander. The output of the second pass in turn led into an assembler which handled the specific formatting for the machine. Later a third pass was added intermediate between the first two in an attempt to optimize the object code. The quick-and-dirtiness came through when the original language subset specs had only a single diagnostic, namely, ERROR. That has been expanded so that maybe now there are half a dozen, but the only help you get is that the message appears in the neighborhood of the statement that caused the trouble. The compile rate, which was never a major issue, turned out to be a few statements per second. It has been improved a little with time, but, more critically, the object code that is generated has improved to a respectable 10 instructions per executable statement. (There's obviously a large variance attached to these figures.)

The environment that the EPL compiler had to fit into is significant. First of all, we had adopted as a machine standard the full ASCII character set of 95 graphics plus control characters, so one of our first projects was trying to map a relationship with EBCDIC—the IBM standard. We also intended to use the language in a machine with program segmentation hardware in which programs can refer to other sections of programs by name. Fortunately, we could use the $ sign as a delimiter to allow us to have two-component names. We also expected the compiler to generate pure procedure code which was capable of being shared by several users, each with their own data section, who might be simultaneously trying to execute the same procedure. We also wanted to establish as a normal standard, although not a required one, the use of recursive procedures by means of a stack for the call, save, and return sequence linkage information and automatic temporary storage. We also wanted to allow the machine to have a feature which we've called "dynamic loading" in the sense that an entire program isn't loaded per se; the first procedure is started, and as it calls on other procedures, these procedure in turn are automatically fetched by the supervisor on an as-needed basis rather than on a pre-request basis. This, of course, is in conflict with any language which allows storage to be pre-declared by the INITIAL specification within any possible module that is ever used by the program. (This problem also comes up in FORTRAN.) We

also had a feature in the machine which we called "segment addressing", which is such that when you want to talk about a data segment you don't have to read it in through input/output; rather, you merely reference it and the supervisor gets it for you through the file system. In other words, we were trying to design a host system capable of supporting software constructs which make it easier for people to write software subsystems. In this rather sophisticated environment, one of the problems was that much of the time was spent finishing the design of the compiler so as to implement the mating of the language constructs with the environment. The things that caused trouble were the SIGNAL and ON conditions, which are relatively trickly ideas and which clash head on with faults and interrupts. The call, save, and return conventions had to be mated into the standards of the system. Problems of non-local GO TO's and the releasing of temporary storage which has been invoked had to be licked. Most of these problems are implications of the language if one thinks it through, for the language has a lot of assumptions in it about what kind of an environment it is going to be in. There are also little subtleties, like when you're talking about strings of characters and operators, what is the role of control characters, i.e., codes without graphic representation such as backspace, when encountered in strings? There are also obvious difficulties in that the language doesn't discuss any protection mechanisms, a feature that every system mush have to implement a supervisor-user relationship. Thus, there needed to be some additional modifications made to the compiler to make that work out. And then there are strategy problems within the implementation, such as how you're going to implement internal blocks and internal functions. These also took some time to work out and were one of the principal reasons why the compiler implementation was slow going. Further, it was done simultaneously and in parallel with the system design. I would say with hindsight that we didn't put enough effort into trying to coordinate the two. The reason we did not was that, to a first approximation, we felt that the language was a decoupleable project. That was a useful thing in the early days, but as we came home toward the finish line in the design, it began to haunt us that we hadn't worked out some of these interface ideas more carefully, and we had to pay the price of redesign in various parts.

One preliminary conclusion we draw from the experience is that PL/I went too far in specifying the exact environment. There are a lot of ideas that should be subroutines and not part of the language. I don't mean they shouldn't be thought through, but to think them through is not the same as putting them in the syntax of the compiler. In particular, things like SIGNAL and ON conditions could indeed be implemented as subroutine calls and be part of the environment of the host system. I don't think they belong in the language per se, although if one makes the language embrace a standard subroutine library, then I of course agree.

I'll say very little about the vendor's compiler. They estimated it would take 12 to 18 months. After approximately 24 months, we stopped expecting anything useful to appear. One of the principal reasons they failed was that there was a gross underestimation of the work, by a factor of three to five, and it was impossible to mount a larger effort by the time the underestimation became evident. Thus, the pioneering EPL has become the standard system-programming compiler.

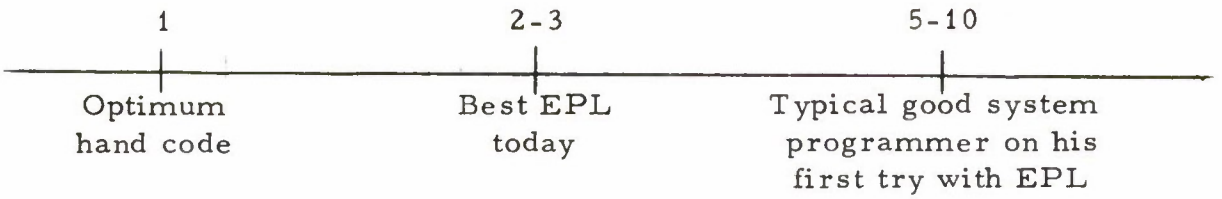
Let me next talk about the use we made of the PL/I language. A strong point, we felt, is the ability to use long names which were more descriptive. People still get cryptic, but they're not nearly as cryptic as they were. The full ASCII character set is a strong point because we wanted to deal with a well engineered human interface. The structures and the data types, as I mentioned earlier, we consider to be one of the strongest assets (this perhaps comes as no surprise to COBOL users, but this feature is very important when you're trying to design data bases). The POINTER variable and based storage concept, along with ALLOCATE and FREE, have been pivotal and crucial and have been used extensively. Some of the features like SIGNAL and ON conditions, which have cost us a lot of grief, at least in principle have been very graceful ways of smoothly and uniformly handling the overflow conditions and the like which suddenly trap you down into the guts of the supervisor. In previous systems we have always had the quandary of how to allow the user to supply his own condition handlers in a convenient way. We're not sure that the price is perhaps too high, but the mechanism does look good. The SIGNAL mechanism also is an elegant way of handling error messages. One of the problems with an error, when it is detected at a given subroutine level, is that its significance isn't always understood. For example, the square root routine may encounter a negative argument but only the subroutine that called it knows the significance. Maybe it means that the cubic equation solver has three roots that are not all real, but that again isn't the true significance. In fact, it may mean that the experimental data that was being analyzed was merely noisy and incorrect and that this data point should be abandoned. The SIGNAL mechanism allows each subroutine in the line to insert a message if it is wanted and allows the square root routine, that didn't know who called it, let control go back in the right way. Finally, a last point about PL/I that is perhaps obvious, is that the conditional statements that are straight out of ALGOL are very valuable.

Overall, the general result that we got from using PL/I was a rather small number of programming errors (after a programmer learns the ropes), in fact a sufficiently small number that one of our major sources of trouble is that a lot of bugs have been caused by mismatched declarations, getting parameters in a calling sequence inverted, getting argument types in calls mixed up—all clerical errors in which the language gives you no help and our implementation

doesn't either. In fact, this is a defect in the language in the sense that the independence of the separate compilations has left a gap in the checking of types. (Sometimes programmers have used mismatched declarations for gimmicky convenience or efficiency, although we have tried to avoid it because it obviously destroys machine independence.) We also found that skillful system programmers who know the machine well don't want to work in machine language because they make too many mistakes. This condition is aggravated because in modifying the machine we retrofitted a lot of involved ideas onto a somewhat ornate order code. Regardless of the reason, however, we find that programmers would rather get things done than twiddle bits.

Another major effect of the use of PL/I has been that so far we have been able to make three major strategy changes which are really vast redesigns. One of them was in the management of the high-speed drum that did most of the paging. It was reworked, quite a while ago, when some insight developed which allowed a tremendous amount of bookkeeping to be eliminated. The amount of code that was involved dropped from 50,000 words to 10,000 words. This total rework was done in less than a month (although not completely checked out because the person wasn't working full time on it). A second redesign occurred in the area of a special high-strung I/O controller which has all kinds of conventions and specialized aspects. The first cut of the control program design was a little rich; it ended up involving around 65,000 words of code. After people finished debugging it and recovered their breath, they took a closer look at it and saw that, by cutting out maybe 10% of the features and changing some of the interfaces and specifications, they could streamline it. Two good men working very hard did the reworking in less than two months. The two months were peak effort, but they did do it. The program basically shrunk in half, down to 30,000 words, and it runs about 5 or 10 times faster in key places. This kind of redesign is invaluable. It gives one the mobility that one is after. It may be true for the use of nearly any compiler—I'm not trying to argue that this is exclusively a PL/I attribute—but this is the experience we're getting. Finally, we made another major change in the system strategy of handling own-data sections, which we call "linkage sections". We were keeping them as individual segments, but we reorganized things so that they were all combined into a single segment because some of our initial design assumptions had not been correct. This reworking was done in the period of a month. The change was serial to the main line of the project development, so that it was a rather important period of time to minimize.

Now, there's another side of the coin, namely, object code performance. This aspect is illustrated in the diagram on the next page. These



figures represent only a preliminary view. The diagram illustrates the object code execution time in the horizontal direction. (It could also be object code space since, roughly speaking, they are similar.) The unit 1 represents optimum machine code or hand code, where one uses all the features of the machine but stays within the specifications. The next point on the figure represents the best results obtained to date by careful juggling and tuning of EPL written programs. At the moment the comparison to hand code seems in the vicinity of 2 to 3 times worse and is largely because the compiler cannot optimize very well. A lot of redundant expressions are being calculated; this is especially true with based storage and pointers, where it is easy to build up fairly elaborate expressions to access a variable and then at the next occurrence repeat the calculation. Finally, and this is perhaps the one shocking note that should be taken with some caution, we find that a typical good system programmer on his first try produces EPL-generated object code which is perhaps 5 to 10 times as poor as hand code. I think this is the main problem with PL/I, because a factor of 5 or 10 at the wrong places can sink the system. The reason for the factor of 5 or 10 seems to be principally that programmers don't always realize the mechanisms they are triggering off when they write something down. The usual pattern when one writes a program is to think of four or five ways that one can write out a part of an algorithm and to pick one of them on the basis of knowing which way works out best. What has happened is that people are too detached. For example, if you use a 1-bit string for a Boolean variable, it turns out in our particular implementation you generate a lot more machinery than if you'd used a fixed integer. Similarly, varying strings carry a fairly stiff pricetag in our present implementation (although ways are known to improve matters a little), and they must be used with caution. Occasionally, too, we've had mishaps where the machine independence works against us in the sense that a man declares an array of repeating 37-bit elements and the compiler dutifully does it, straddling word boundaries mercilessly. The best we've been able to do so far is to get the compiler to at least remark "IDIOTIC" on the object code listing. There may be other reasons for the factor of 5 to 10, such as the language learning time, but we do not consider them important. Issues such as the 10-to-1 variation in ability among programmers of similar

experience (discussed in an article by Sackman, Erickson and Grant in the January 1968 Communications) I think can be discounted in our case. Our technical management has been thin, but we have kept careful track of the individual programmers so that mismatches with work assignments have been minimized.

With regard to remedial measures for upgrading the system, one must remember that most of the code in the supervisor doesn't matter (it's not being used most of the time), so the key thing is program strategy. I don't have time to discuss here how we localize the parts of the code which are the functionally important parts, but a segmented machine pays off handsomely. Meanwhile, we are learning tradeoffs between the different supervisor mechanisms. In addition, we are trying to develop checklists of things to avoid in the language. It turns out to be rather hard to get people to generalize, so it is slow going. On a long-range basis GE is developing plans for an optimizing compiler, but it isn't going to help us right away. On a preliminary basis we are also studying smaller subsets of PL/I with perhaps modifications and changes to the language so that the implementation is more uniformly potent—or impotent, depending on how you look at it. That is, the user would be constrained to a language which would implement well regardless of whether he takes one choice or another. And, finally, there are some coding tricks that might have helped if we had thought of them sooner.

One of the key problems in our use of PL/I has been that the programmer doen't have feedback. If the compiler gave the programmer, say, a time and space estimate on each statement that he writes, and if he were in an interactive environment developing the program (i.e., he could get quick return on his compilations), he might be able to form some intuition about what he's doing. To implement such estimates is not a trivial problem because a lot of the mechanisms that are invoked are shared, so that there needs to be a way of designating the shared mechanisms and showing why they are included.

Finally, as a last resort in improving supervisor performance we can always go to machine language on any supercritical module. But this isn't a panacea, because it is easy to be swamped if one tries to put too much in machine language and, moreover, one has lost mobility. I always consider going to machine language to be like parachuting out of an airplane.

I have a few general conclusions. I think that in the language area there has been considerable leap-frogging. FORTRAN was the first compiler with any

widespread use, and it suffered because it wasn't systematic to implement and was somewhat clumsy to use. It was, however, a practical language. ALGOL was in a sense a reaction, but it suffered because it left out the environment and didn't come to grips very squarely with the implementation. PL/I in effect is a reaction against ALGOL's not having considered the environment, but it suffers from being designed without well formed plans for a systematic implementation. The notation of "systematic" is important; without it the cost of implementation, the speed of the compiler, and the quality of the object code may be off by factors of ten or a hundred. Nevertheless, I admire the PL/I design effort and consider it valuable because it has inspired language experts to try harder; in effect it has set as goals what is wanted. The fact that the language has not been implemented well by anyone I consider to be an object lesson. Nevertheless, techniques for mastering the problems are being found. In addition, people are beginning to think of ways of accomplishing the same functional characteristics without the same internal problems. One of the ways is to try to minimize the language syntax and to think through more carefully what is the subroutine library.

Future languages will come and they'll be beneficial, too. But PL/I is here now and the alternatives are still untested. Furthermore, I think it is clear that our EPL implementation is going to squeak by, and in the long run the Multics project will be ahead because of having used it rather than one of the older languages. Now, finally, the last question, which I think is a tough one: If we had to do it all over again would we have done the same thing? I'm not totally sure of my answer; I just don't know. We certainly would have designed the language more carefully as part of the system; that was something we didn't pay enough attention to. If it was EPL or a PL/I again we would have tried to strip it down more. With hindsight we would have modified it to some extent to make sure that it could have been implemented well. If it was another language we would have tried to beef it up with things such as are in PL/I, and maybe modify it. Either course of action takes a lot of design time, and that's the dilemma: in effect, you want your cake and you want to eat it too. I think the decision probably hinges on whether or not one is trying to meet a deadline. I would probably use FORTRAN to meet a firm deadline. But if I'm trying to solve a problem with a future, I think I would use either PL/I or its functional equivalent—and the choice will have to be answered in the future.

PL/I AS A TOOL FOR SYSTEM PROGRAMMING

Questions and Answers

Question: You talked about redesign as being a major part of your development, where you got a big payoff.... Could you comment on this... maybe your opinion on how well you should design to start with and how much emphasis you should place on redesign?

Answer: We consider our design documents in absolute terms to be mediocre, but in relative terms to be good compared to other programming efforts we know about. We worked hard on this, and we did it for a reason: one of our principal goals was to be understood. It's also saved the project over and over again because people can see what is going on, and new people can join the project. These design documents weren't just written and accepted; they were usually bounced around, and the first proposal wasn't usually the final one. There was a deliberate self-criticism in the design; the goal that was being sought was functional isolation of different ideas. We felt strongly that one should try to design as carefully as possible before writing programs. Debugging incorrect ideas is a very expensive waste; nevertheless it is almost impossible today to correctly foresee all the implications of a large system design. Thus some redesign is inevitable, and it is here that the compiler language speeds the process. In particular: 1) you don't get distracted by bit-twiddling, and 2) you can see what you're doing and don't get lost in a sea of details. It allows you to keep focused on what you're trying to accomplish rather than getting caught in tedium.

Q: I have a question to ask about your comments regarding systematic implementation. Do you feel that as PL/I is implemented by a variety of manufacturers, the implementation will effectively reduce the effectiveness of standardization?

A: Are you worried about the question of whether the different manufacturers will create different languages—is that what you're saying?

Q: Well, I'm suggesting that when the language is implemented by someone, his implementation will vary sufficiently from someone else's implementation to cause one to lose the advantages of having standard syntax.

A: I don't think so. Even FORTRANs aren't the same in most machines. The minute you change the word length underneath FORTRAN you normally change many of the subtleties, including precision and the like. I think the best one ever ought to expect out of the present view of so-called "language standards" is a kind of a first cut at having an approximate version of the program that will work on another machine. You still have to audit and edit accordingly, and this is, I think, the best we could hope for out of PL/I. In the case of PL/I, however, I would expect there to be a larger class of programs for which one would be indifferent to the subtle variations and only worry about cases of drastic differences in behavior.

Q: Does that apply also to, say, a program's dependency on an operating system?

A: I think that in this area the discrepancies between PL/I implementations will be large. If you get totally wrapped up with an operating system which is peculiar to a particular set of hardware, you're trapped. You're just going to have to rework it on another machine. It is a long-range goal, as yet unachieved, to minimize this problem. We've considered the problem some because one of the obvious questions that arises when you have a system which has been largely implemented in PL/I is: Could you put it on another machine? The answer is "Yes," although I think it's still at the level of a "technical challenge" even with similar hardware. To do it one would have to go through and modify and edit all the programs to make it come out just right. There are also some strategy changes probably required unless one actually built an identical machine. Nevertheless, such a task is still preferable to having to start all over, writing off as a total loss the ideas and efforts of several years; this is the alternative to working in a compiler language for system programming. I don't really see how there's going to be any progress in the field until we stop killing off our system children.

Q: In light of the fact that you bemoan the lack of systemization and I get the impression perhaps it's not quite time to standardize and so forth with PL/I as it is, at the same time it strikes me from your comments that you probably, you in your project,

know something more about machine characteristics that are suitable for or unsuitable for a language like PL/I if not PL/I itself. Do you have any comments to make there? That is, rather than carrying PL/I to another machine which is again incompatible, what about designing a machine which would make PL/I a useful tool, a more useful tool?

A: Well, I think it's useful now. I don't think one has to apologize for the language.

Q: I mean more useful, I mean more ideal.

A: The fact that we've gotten down to a level of only two or three times clumsier than hand code is perfectly good enough for many applications. The problem at the moment is that you can stray off the path; it's like skiing down a mountain and going off the trail into the woods unexpectedly. Some of the problems are intrinsic complications of the language, and to some extent it has to be streamlined to do a much better job. At present this issue exceeds in effect any hardware improvements to favor PL/I. As far as being accepted as an industry standard, I guess I'm a little more laissez-faire about this than most people. My own reaction is that one can judge for himself when one has a de facto standard, and treat it accordingly. It already is a de facto standard of some sort, and there will be future language standards. But they obviously will have to compete with PL/I and show that they do, at least in some sense, a better job.

Q: You indicated that the project started out as a two-year plan. I was wondering if you'd comment on whether much of your schedule was perhaps influenced by hardware problems in addition, say, to the compiler implementation problems. What were your principal problems? And, talking about doing it over again, do you expect that you could, looking back and starting fresh and assuming no uncontrollable factors, have done it in the two-year period? Is it just this typical problem of underestimating the complexity of developing a major software system?

A: I don't think we're embarrased that it went from two to four years; that's sort of par for the course for a research project,

and that's what it turned out to be. If one really wanted to predict performance or schedules he'd have to do something he had already done before. That's just what we didn't want to do. If we wanted to meet a two-year deadline, we would have had to say, "Imitate CTSS. Copy it slavishly." If we had done this, though, we wouldn't have increased our understanding of computer utilities and we would have propagated many system design limitations. I think one really has to face up to the fact that if you're going to try something new, whether it be a language or a system, you had better leave yourself some slack. A factor of two is pretty routine on research programming projects. We were facing three major problems all at once: new language, new hardware, and new operating system, not to mention the fact that we had three organizations involved and in three geographical locations.

EXPERIENCES IN PL/I AT IMPERIAL CHEMICAL INDUSTRIES, LTD.

J. Michael Sykes
Manager, Programming Services
Management Services Department
Imperial Chemical Industries, Ltd.

I think I should start by way of introduction by telling you just a little bit about what ICI is so that you can get some idea of the perspective of the environment in which I am speaking. We are one of the largest U.K. companies with the total of something in the region of 100,000 employees. That may not be very big by U.S. standards, but it is pretty big by U.K. standards. It's a highly decentralized company with nine operating divisions which, apart from their responsibility to make a profit and follow general company policy, operate more or less independently. That is to say, they make their own decisions about what equipment they get for any purpose, in particular for computing. The effect of this is perhaps a slightly untidy situation. At the time when 360 was announced, we had a wide variety of computers; in fact, our computing capacity was more or less on a "you name it, we have it" basis. This was partly deliberate in the sense that when computers first began to be installed in the U.K. around about 11 years ago, it didn't seem a good idea to standardize on one manufacturer. As time went on, there was more and more talk about standardization (and more and more talk about what it meant), and eventually the head office did achieve a fairly substantial policy decision by putting the company's name down for eight 360s on the day the product was announced. From that point on, you might suppose that everything has been standard; but it hasn't. Shortly after that time, the first technical report on PL/I appeared. We didn't take too much notice of it at the time, but a few months later when we started hearing things about what PL/I was, we became interested because it was evidently the intention to replace FORTRAN, COBOL, ALGOL, and pretty well anything else you care to think of.

Before I go on to say what we did about it, let me just mention we have at the current time nine 360 computers in seven installations, ranging from two Model 50s down to a couple of Model 30s, one of which is running with a 50; thus we are in quite a big way in computing in the U.K., and the experience at these different installations is very diverse, as you can imagine. Another difference I should emphasize between the United States and the United Kingdom is that labor, and programming labor in particular, is very much cheaper in England than here, and the hardware cost is about the same. This means that the programming costs are liable to be about half in U.K. of what they are here for a comparable amount of programming, which very much affects our attitude toward

high-level languages. So it wasn't until the 360 came along that anybody in ICI gave very much thought to high-level languages, other than for open shop, or amateur, programming, where they were fairly well established. For data processing, there was very little use of COBOL before 360. It was because there was so little use of COBOL, and I happened to be involved in what use there was, that I became the COBOL expert and subsequently the PL/I expert. We have the traditional method for appointing experts: you find somebody who has read the manual and written a program and he becomes the expert.

Just a little about myself. I started programming 11 years ago on a machine called a Ferranti Pegasus, a very small machine by today's standards, comparable in power to the IBM 650 and the same vintage. We found programs on these machines very slow to write because they had to be in machine code; we hadn't even got a symbolic assembler. So when we started thinking about more ambitious projects, specifically a real-time system for processing customers' orders, allocating goods from stock, and notifying the warehouse what they were to send to the customer, we started thinking about bigger computers and high-level languages, since we expected that there would be a tremendous amount of program in this system. Moreover, such a system would never be complete—it would always have to be altered. In any case we weren't sure that some of it could be programmed at all so as to run economically. That is to say, if you have to process a thousand orders in the working day and it takes you one minute to process each one, you're just not in business. So, we had to do some evaluation work and this we did in COBOL. I first used COBOL on an RCA 501, which went in England under the pseudonym of the English Electric KDP 10. This was one of the first implementations of COBOL 60, and there were no paragraph names and all GO TO's had to be to statement numbers. Later, when the 360 was ordered we transferred this work to a 7094 at an IBM Data Center and discovered that even though one uses a standard language, it doesn't necessarily mean that one can transfer from one computer to another without any trouble. We had a similar experience when programs which had been developed on the 7094 were subsequently transferred to a Model 360. Even with one manufacturer, we found a lot of incompatibilities between one implementation of COBOL and another.

Now to get back to PL/I. As you all know, the report on the New Programming Language, NPL, came out in the spring of 1964. It was about nine months later that ICI set up a working party. This was done by collecting two proponents of each of the languages that PL/I seemed to be competing with. Thus we had two representatives of ALGOL, two

of FORTRAN, two of COBOL (one of which was myself), and two of K-Autocode, a language nowadays almost exclusive to ICI and roughly equivalent in scope to FORTRAN. This working party met three times only. It didn't need to meet any more because at the end of that time it was able to reach a unanimous verdict. I think it will be relevant if I quote a little from the report, which may seem fairly old but has by no means lost its relevance.

> "PL/I has a much wider range of facilities than any of the other languages under consideration——namely FORTRAN, COBOL, ALGOL, and K-Autocode."

To produce the evidence for this conclusion we did a fairly detailed comparison of PL/I with each of the other languages, except that the two who were enjoined to compare PL/I and FORTRAN really didn't think it was worth the trouble. They said, "On practically any facility you care to mention, PL/I is better; so please let's forget FORTRAN." In any case, there wasn't much use of FORTRAN in ICI at that time, and the people who were using it didn't think too highly of it in comparison with K-Autocode because the latter was giving them very much better run time diagnostics. There was also a rider to the report:

> "We have not included the details of the PL/I : FORTRAN and PL/I : K-Autocode studies since the points which arose are largely covered by the PL/I : ALGOL study. There are a few facilities in FORTRAN and K-Autocode which are not available in ALGOL which deserve separate consideration. COBOL needs separate consideration as a data-processing language so it comes in a different category."

One of the things we had been asked to comment on was the desirability of having a single language for both mathematical computing and commercial data processing. We decided that this was a good idea. Precisely what it's worth, of course, is a matter on which we couldn't comment. What we did say was,

> "If it is achievable, then it is a good thing."

We quoted from the NPL student text, which was already out:

> " 'Linear programming, decision making, forecasting increase the scope of computations and hence the area of competence of the business programmer. Scientific programming, on the other hand, is faced with the task of handling problems with masses of data with increasing orders of magnitude and therefore forcing us to look at the input/output requirements. The differences between scientific and business programming are thus becoming less distinct.' "

I think this is generally accepted. I have certainly found it to be the case in my shop.

> "The working party finds that PL/I is a suitable contender. Footnote: At present, it is the only contender."

It is the only language which has attempted to combine the virtues of pretty well the whole field, although it will probably not be long before some other claimant appears, e.g., ALGOL 60 could, although I believe ALGOL 68 is just about around the corner; but how it's going to compare to PL/I, I really couldn't say, because I can't see around corners.

We were also asked to comment on such things as flexibility and elegance of the language and its open-endedness. We decided that:

> "PL/I is as flexible in application and as open ended in design as is likely to be the case in any current programming language."

I would add that since some of the features originally suggested have subsequently been dropped from the language, it is even more open ended than it was. Hardware dependence was another topic to be considered:

> "While there are some features of PL/I which are hardware dependent, the main problem in transferring PL/I to another system of machines would be the sheer programming effort required to write the compilers. It is reported in Datamation that IBM is investing 600 man-years in this project."

EXPERIENCES IN PL/I AT IMPERIAL CHEMICAL INDUSTRIES, LTD.

A recent estimate given me quite informally was that perhaps 200 man-years had actually gone into the compiler so far. It is still an awful lot, so this is obviously a problem for any manufacturer who is thinking of implementing it. We commented on this:

> "While much of the effort is doubtless concerned with documentation, a project of this magnitude could hardly be undertaken by the relatively small software groups with which we are familiar in this country."

This is a reference to the fact that in England you tend to get half a dozen people together to write a compiler rather than half a hundred.

Logical simplicity and elegance was another factor we were asked to comment on:

> "While PL/I, because of its wide range of features, may not be as logically symmetric or compact as might be desired, there would seem to be no intrinsic difficulties in teaching the language."

I'll comment on teaching later. It is obviously very closely bound up with the elegance or logical simplicity of the language. Another comment we made on this point was that:

> "The ease with which the amateur can use the language [here we mean the open shop programmer, using it, please note, as opposed to learning it] depends on the efficiency of the diagnostics, and some of us have some misgivings on this matter."

I think this is because we were very conscious of the fact that IBM had not been accustomed to providing such effective, helpful run-time diagnostics as we were accustomed to having from our own K-Autocode (in particular, a source language dump). An interesting sidelight on the question of logical simplicity was that we could not apply the criterion that used to be applied: a good programming language is one you can specify on two sides of a sheet of foolscap, or better still, on one. The chairman of the working party, R.A. Brooker of Manchester University, suggested that another possible criterion one might apply to a programming language was to take the number of facilities and divide this by the number of pages in the manual, but we didn't know how to start counting facilities so we had to drop this idea.

We did come up with a quite unequivocal conclusion:

> "PL/I should be used as the preferred language for data processing and mathematical computing, if and when the compilers are shown to be satisfactory."

Now we didn't say what we meant by "satisfactory", but in the absence of any compiler, this is obviously something that you can't take for granted. A rather nice quote that one of the members, not myself, provided to this report was that:

> "We should, as a company, move towards PL/I with deliberate but cautious speed."

That about sums up the report. As the man who had to do the COBOL comparison, my conclusion was that there wasn't really much to compare between PL/I and COBOL; PL/I had it. There were one or two COBOL features that weren't in PL/I. We felt that either they would be put in——I am thinking specifically of the numeric-alphabetic class test and the EXAMINE feature, which can be quite useful——we thought either these might be added, or if they weren't, you could probably handle them with assembler subroutines of a fairly general-purpose sort. Although we did study the specification of the COBOL report writer feature, which was one of the most significant things absent from PL/I, we didn't, in fact, get to the point of understanding it sufficiently well to be able to say what it was worth in practice. We felt that it was a rather clumsy facility and that we would be hardly worse off without it.

Now we come to the question of experience. We first got a compiler in a prerelease version in July 1966, and we started playing around with it straight away, on an experimental basis. As a matter of fact, some of the more enthusiastic people for PL/I were technical users who were interested particularly in the new facilities which were becoming available, and they got themselves into difficulties because they used facilities that they didn't properly understand in ways that evidently hadn't been tested very thoroughly and that didn't always work. But we did, by doing a fair amount of experimentation, reach a decision in September 1966 that the shop of which I was then in charge would go over wholly to PL/I fairly quickly. It was convenient to do this because we were at that time transferring a number of quite small schemes onto 360 (when I say "small" I mean a scheme which consists of three or four programs of 200 to 300 statements each) from our Pegasus computer, which was still operational at the time. This provided a very convenient environment

for evaluating the new language. However, at the same time we were concerned to tackle something a little more substantial. Now, I don't really know how big this will appear to be to you by your standards, but it seemed fairly sizable to us. I said before that ICI was very decentralized. This meant, in particular, that each separate division has its own responsibility for paying the salaries of the people who work for it. It was suggested that on 360, now that we all had the same type of computer, it would be nice if a team, preferably in a central department, were to provide a set of programs which would enable each installation to run its own salaries project without having to write its own programs. In September 1966 this scheme had been fairly well specified and it was decided that the ground was safe enough to program in PL/I. It is the only scheme I'm going to talk about because I was, and still am, very closely associated with it. But, round about the same time, we did elsewhere in the company start on two other schemes, one of which is by now operational, the other nearly so.

We set up a team of six programmers plus a team leader, and they started work in September 1966 with the intention of going live in April 1967. In fact, we didn't get into production till June, largely because of the difficulties I'm going to talk about. In fact, the team was down to about three by the end ot the period because we found that once the coding was done, we could reduce the size of the team for the system testing. It's easy to get two programmers to write one program, but when it comes to the system testing, they tend to get in each other's way.

There were a number of programs in this project, a large number of small ones, as you would expect, particularly for doing things that only have to be done once a year, but the main programs were basically three. There is a program for checking the input, which is read in random sequence, and when it has been checked and sorted, another program updates the master file. The third program then processes every record and calculates salary, pension contributions, tax, and all the other things that are required.

I shall not refer further to the first program except to say that we did discover when writing it that the date-checking facilities of PL/I were somewhat inadequate. Thus, for example, having read a card and knowing that the whole of the information on that card should be numeric, one has no convenient way in PL/I to check that it is, whereas in COBOL there is a good way to do so. I believe this deficiency is likely to be made good, so there's no need to dwell on it.

The second program, the file update, was a pretty large program by our standards. It came out at about 7,500 PL/I statements. There was a certain amount of duplication in here because it consists of about 60 modules of program, and various data declarations appear in every module. We had quite a lot of trouble with this program the first time we linkage-edited it. We intended to run the whole system on a 128K machine, so we reckoned it would have a partition of about 105K to 110K to run in. The first time we linkage-edited the update program with as much overlay as we thought we could manage, we found that we had a load module of about 140K. So, we had to go back and think again about dividing up modules into smaller modules and reorganizing our overlay structure, which was a complicated exercise because, as I have indicated, we finished up with about 60 overlay segments. There were at that time problems with overlaying in PL/I, but, in fact, this did not delay us seriously because we wrote our own program to read the PL/I object decks and remove the cards that caused the trouble. However, we found that we were still using a lot of core.

It wasn't until this time that we actually started looking at object code to see how the compiler came to be using all this core, and, although I accept that we are not really here today to talk about the implementation, we did learn one or two things at this time which made us realize that it isn't necessarily all the compiler's fault when it produces inefficient object code. This is very important. I am sure that it has been true of other high-level languages before PL/I, but it is very much more true of PL/I than it was of any predecessor. We found, for example, that the quantity of library subroutines we were having to call in, as a result of the way we had written the program, was anything up to about 35K. It turns out that to do nothing you need about 6K, but by using a wide variety of facilities, and perhaps using them rather untidily, we found we were using as much as 35K. At this point there was too little time to review everything we had done, but we did do a certain amount where we knew there were worthwhile gains to be made. This quantity of subroutines is almost implied by the nature of the language.

Another thing we found was taking up a considerable amount of space was dope vectors. And, if you think about it, these are implied——or some information of this kind is implied——by the language itself, because as long as a subroutine is allowed to declare a parameter as being a character string of indefinite length or an array of indefinite bounds, then the information about the length of the string or the bounds of the array has to be passed at call time. This is quite fundamental, and the information

is passed in the form of dope vectors. We found that our salary master record, that is, the record which is needed to pay a man's salary every month, requires 690 bytes of core, and when it's written out it is 690 bytes on tape, but its dope vector while it's in core occupies about 800 bytes, which we think rather excessive — but I don't know what can be done about it. We also found that quite a lot of core was taken up by code extracting information from these dope vectors. I don't know what can be done about that either.

The ability of any PL/I implementation to acquire core dynamically is, I accept, a very desirable feature in some applications, but it got us into trouble because we knew about dynamic storage. We knew what it meant so we decided not to use it — explicitly. We didn't declare anything as being automatic or controlled, but we found that when we entered into a subroutine, core was nevertheless acquired dynamically for saving registers and for temporary work space, so that it wasn't good enough to get our load module down to the size of our partition merely; we had to reduce it to a lot less than that so as to leave room for storage to be acquired dynamically. We also found that if an error condition occurred during execution and there is an ON-unit for that condition, core is acquired dynamically at this point in fairly substantial quantities (about 4 or 5K). Thus for a time we had to test the program, executing in the core available, but if an error occurred and the error-handler was entered, we ran out of core, which made discovering the cause of the error a little tricky.

We hadn't intended ever to get to the point where we needed to trace through PL/I dumps, but we found that there were situations in which we were forced to. In particular this happens in situations of the sort that Professor Corbató referred to this morning, where there is a mismatch between the attributes or number of arguments being passed to a procedure and the way the procedure has declared the parameters it expects to receive. The compiler doesn't detect this where both procedures are external, and the consequences can be disastrous, particularly, for example, where the procedure thinks it is picking up an address, and what it is actually picking up is a small binary integer. After this scheme had been in production for several months, one of the installations using it went to a release of OS/360 which had storage protect for the first time, and they got a protection violation. It turned out we were using a location in core with a very low address as a switch, which is hardly a very good idea. This was due to a mismatch of parameters, and we had some trouble locating it. Similarly, if a record is declared as external because it has to be known to all the procedures, but the declaration is wrong in one of these procedures, the results are quite unpredictable.

Now, I think we have to accept that this is one of the prices we have to pay for modularity. The compiler doesn't know what other routines are going to be linkage-edited with the one which it is currently compiling (unless some means is found in the future of providing it with this information). I think it would be unwise to ask the compiler to build in code that would check these definitions at execution time. Certainly we do not want this sort of code included for every execution, although it might be useful to have it during debugging, so that at least you can establish that you have a working program before you say, "OK, now I'm ready to do without these facilities. Give me some more efficient code."

We got some nasty surprises in other directions as well. It is often necessary when updating a file to insert a new item into it, and in such a case there is usually in core the previous record, which you don't want to use at the moment. It has to be put on one side while the new record is created. This shunting out to one side may be done by a simple assignment statement to move the 690 bytes I referred to previously; the assignment statement to move these 690 bytes generated about 5K of in-line code. Now, if you look at the definition of the PL/I language, this should come as no surprise. Admittedly, the compiler might have been able to detect that there were no conversions required, that the mapping of the source structure was exactly the same as that of the target, but it didn't. This is not difficult to avoid; it is just something you have to know about. Another statement we had in there which we thought was extremely valuable for debugging purposes was PUT DATA (Salary Master Record). This generated about 7K of code, which is expensive. However, we didn't cut it out and throw it away; we cut it out and overlaid it, and we still think it's a very valuable statement. It does get executed occasionally when something goes wrong or when a record is deleted from the file. We don't want it out of the language; we don't even care if it is not more efficiently implemented; after all, it is likely to be inefficient because of the necessity of having a symbol table and data element descriptors; but it is something one must be aware of and something one must be prepared to overlay. We overlaid not only the module that called it, but also the PL/I library subroutines that it needed.

Another thing that I was rather unhappy about, having been used to something different in COBOL, was that constant subscripts are not evaluated until run time; for example, if you refer to the fifth element of an array, it doesn't calculate the offset of the fifth element from the beginning of the array at the time of compilation. In some cases it could do this, but suppose a two-dimensional array is passed as parameter; the subroutine called doesn't know what the bounds are at compile time, so

clearly no offset would have been compiled in however constant the subscript.

There are yet other fundamental inefficiencies in the language associated with stream I/O, which is a facility that I really wasn't accustomed to, not having had much FORTRAN experience. Any implementation is likely to be inefficient in this area because the compiler doesn't know what conversions are going to be needed until the statement is executed; this occurs because the correspondence between the data item and the format item is not resolved until execution time. How can this inefficiency be avoided? The program has to build up lines of print in core, much as is done in COBOL, and then use RECORD-type output. Furthermore, we hadn't realized that if list-directed input is used, this isn't going to be very efficient in any implementation. It can be very convenient and there are times when it's a good idea to use it. But in a data-processing system I don't think there will be very many such occasions, because this statement has to be prepared to accept any valid constant in the input stream and consequently has to have available whatever subroutines may be necessary in order to convert this constant into the appropriate internal form. Programmers need to know such things. Again, I am reminded of what Professor Corbató had to say about ON-units. If you rely too much on them, for example for replacing leading zeros in a field being read to a picture format of all 9s, they can be extremely expensive. This occurs because whenever an ON-condition arises, the program has to acquire extra library work space and store registers, do one or two other things, which takes a relatively long time.

Although I am generally in favor of the idea, I do have one qualm about the use of a single language for both scientific and data-processing work: there are times when a conflict in optimization arises. The compiler writer should recognize this and provide the user with the means of controlling which sort of optimization he gets. For example, he may be able to do something in the prologue of a procedure which speeds up the execution of an inner loop, and, in fact, this is done to a limited extent in the existing compiler. But in a commercial situation, you may spend more time executing the relevant parts of the prologue than in executing the statement at which the optimization is directed. This is one of my little hobby horses because I am not sure just how well it is appreciated.

In considering these points I have made about efficiency, I would ask you to bear in mind that I have been talking about experience with early versions of the first compiler for a new language. It is a very large language;

it is a very ambitious language. It has facilities which no other language ever had before, so it is not surprising that there is a lot to be learnt, nor is it surprising that the compiler writers have produced, at any rate in the earlier versions, a not very efficient compiler. But we do, I believe, have a language which can, and in my estimation probably will, replace COBOL, FORTRAN, and possibly ALGOL; I would have a better opinion on that if I knew more about ALGOL 68. It even supplies rudimentary list-processing facilities that have previously been available only in languages such as LISP. It has some genuine real-time features which I won't comment on because I'm not much of a real-time expert. I think we have all the makings here of a tool which is extremely easy to use in the hands of the amateur and quite powerful enough to satisfy the professional programmer.

One of the biggest problems remaining to us, I believe, is the teaching of it. Unless the language is properly taught, we will have on the one hand the amateur being taken nastily by surprise by, for example, the precision rules which I believe have probably never been properly taught and rarely even properly understood; and on the other hand the professional turning out woefully inefficient programs. In fact, I would repeat my remark of an earlier occasion: I firmly believe that a high-level language does not enable one to employ low-level programmers, and I doubt if it ever will. The professional will always need to know what the compiler will produce from the statements he is writing—at least in general terms. The high-level language is not a substitute for programming skill. It provides the programmer with the equivalent of a machine tool instead of a hammer and chisel. And the result is greatly increased productivity.

There are some improvements we need in the language, some of which are already on the way. (I hope that in any further suggested improvements close attention will be paid to the law of least astonishment, as enunciated by R. Wexelblatt of the SHARE PL/I project.)

In conclusion I would mention some features that we think are particularly important. We are very impressed with the compile-time facilities. That doesn't mean we think they are good enough, but we think there is a lot that can be done with them even in their present form. They haven't been nearly sufficiently explored. I have attempted to write a simple program generator, something like an RPG, using the compile-time facilities, and this was neither terribly difficult nor did it require an awful lot of effort. % INCLUDE, of course, is a very valuable feature. It does particularly help one to avoid this mismatch of external data declared in more than one procedure, which is very important. The ON-conditions are extremely useful because without them you can't possibly

fix up after an error. And, at the same time, if you leave them out, should the condition arise, you merely get a message, and these are on the whole very helpful. One feature that gives PL/I a particular advantage over COBOL, and which I feel is often understated, is the absence of reserved words. I don't think the fact that PL/I has no reserved words should encourage people to write such horrible things as "IF IF = THEN THEN" and so on, which is of course perfectly legal, but at the same time you don't want a serious risk of falling flat on your face when a new version of the compiler comes out and there are two or three new reserved words in there. I've had this experience with COBOL, and it can be extremely irritating. The versatility of language and programmers is of considerable value to us, I may say. We often have to move programmers between projects, and the shop I operate ranges from the scientific at one end to the data processing at the other. It seems to me on the whole to be an easy language to write in, a more natural language than COBOL and much more concise, and less cryptic than FORTRAN as long as data names and labels are made meaningful. I think using COBOL is a short way to giving programmers writer's cramp. As far as the run-time debugging facilities are concerned, there are still people in ICI who feel that they're not as good as they are accustomed to, but it seems to me that they are a whole lot better than have ever been provided by COBOL, and certainly better than anything I've seen from FORTRAN.

Questions and Answers

Question: What improvements did you discover in object code efficiency, especially in this payroll system that you wrote, from Version 2 to Version 3?

Answer: Funnily enough, not a great deal. We did a trial over one month, shortly after we got Version 3. We recompiled all our programs and ran them under the new version and duplicated one month's work, mainly in order to see whether we were going to hit any bugs in the new compiler. We did get some timings, but there was a fair amount of ham-handedness in actually timing the programs, and it wasn't really possible to reach any general conclusion except that it looked as though there was about a 10% or 20% improvement in efficiency. We were very interested to find one module which actually came out bigger under Version 3 than under Version 2. This we managed to track down to something that didn't really matter very much, and it was only the one, so who cares?

Q: When you said that you found PL/I easy to use, were you thinking in the context of the good, clean programs that you referred to later when you said that, knowing these things took place, you could modify the program to get around them; or were you thinking of just writing the program and letting it go?

A: I was really thinking of the latter case. Let me give an example. On one occasion, we found that our update program had put some bad data on the master file. We reckoned that to generate correcting data and put it through the whole system would take over three hours. (We had at that time a rather high zero time on the file update program; that is to say, it took a long time just to process the file without having any amending transactions.) So we wrote a fix-up program which merely looked at the items, detected which ones were wrong, corrected them, and reported what it had done. To write this, debug it, and run it in production took about two hours of elapsed time. This is the sort of ease of programming I am thinking of. I don't think it gets very much more difficult if people know how to avoid inefficiencies. They have to pay a little more attention to what they are doing, but I don't think it's really onerous. What they are really doing is restricting themselves to a subset, and the subset is certainly no smaller than, let me say, COBOL 61 (excluding SORT and report writer features). They are really restricting themselves to the more COBOL-like features, in fact.

SUMMARY OF LOGICON'S COMPARATIVE EVALUATION OF PL/I

Raymond J. Rubey
Head, Advanced Systems and Languages Section
Computation and Software Department
Logicon, Inc.
San Pedro, California

Logicon contracted with the Electronic Systems Division (USAF) in August 1967 to evaluate the PL/I language by comparing it with other languages in the solution of specific benchmark problems. The emphasis was on the languages, not on the compilers or operating systems utilized. This paper summarizes the evaluation method used in this experiment and its results. Full particulars are given in ESD-TR-68-150, the final technical report produced under the contract. It is emphasized here that the study was indeed an experiment; and that for different benchmark problems, different programmers, and a different time frame in terms of the implementations of various language compilers, a different set of results might very well be expected.

Method

The study encompassed seven benchmark problems in five application areas, as indicated in Table 1. Each problem was programmed twice by the programmer assigned to it, once in PL/I and once in the comparison language: COBOL, FORTRAN, or JOVIAL. The order of implementation was varied as indicated to minimize the influence that more knowledge of the problem might have on the results of the second implementation.

GROSS PAYROLL, the first of the business problems, is a dollars-and-cents type of problem involving the computation of earnings and tax deductions and the generation of weekly reports by personnel and charge numbers. ALOREP2, the second business problem, is more concerned with business data management. It involves the processing of air cargo data files and their editing and updating based on input information. It also checks the input information for validity and correctness.

The MMI (for Man/Machine Interaction) problem chosen for the interactive programming area involves the development of a simple on-line system in which the user enters equation statements, directives to control their sequence of execution, and other directives to output the results of equation evaluation. The TSME (for Throughput Simulation in a Multiprogramming Environment) problem representing the simulation and gaming area involves

Table 1. Evaluation Scheme

Application Area	Benchmark Problem	Languages		Programmer Experience (years)
		First	Second	
Business	GROSS PAYROLL	COBOL	PL/I	3
Business	ALOREP2	PL/I	COBOL	1
Interactive	MMI	PL/I	FORTRAN	9
Simulation and gaming	TSME	FORTRAN	PL/I	10
Scientific	VIG	PL/I	FORTRAN	4
Data management	SPP-A	JOVIAL	PL/I	6
Data management	SPP-B	PL/I	JOVIAL	6

input and execution queues; the scheduling algorithms, job mixes, and resources required by the jobs may be varied by the user. In the scientific area, the VIG (for Vehicle Impact and Guidance) problem involves computation of a vehicle's terminal state position and requires many matrix and vector operations, the standard trigonometric functions, and various limiting and quantization operations.

In the data management application area, there was actually only one benchmark problem, SPP (for Simulation Post-Processor), but this problem was programmed once each in PL/I and in JOVIAL by two different programmers to gain information about what effects implementation sequence and programmer differences might have on the results. The SPP problem involves the extraction of data from a file, its formatting, and the printing of selected quantities based on input criteria.

Each of the programmers has a bachelor's degree and two have a master's degree, one of them in mathematics and one in information sciences. All are professional programmers. The two programmers assigned to the SPP problem were considered as similar as possible in total years of experience, in their education, and in the nature of their programming backgrounds.

SUMMARY OF LOGICON'S COMPARATIVE EVALUATION OF PL/I

Each benchmark problem programmer remained with his own organizational group; he was not assigned directly to the project team. Naturally each knew that his was an evaluation problem, but his instructions were to proceed just as if it were a normal problem from a customer within the company. A detailed programming specification was provided for each problem so that the programmer needed to devote a minimum amount of time to thinking about just what to solve and what kind of program to develop.

An evaluation study group, consisting of the project head and two project analysts, was established to gather all the quantitative and qualitative data resulting from the benchmark problem implementations and to interpret the results. The members of this group did not try to tell the programmers how to approach their problems, and did not indicate any language features that should be used. It was up to the programmers to use the language features they felt necessary. Assistance was provided, where necessary, by the programmer's own supervision. The members of the evaluation study group were available to help in solving any particular system problems or unique implementation-dependent problems that arose, but not to assist in programming the benchmark problem. To as great an extent as possible, the environment was kept similar to that for typical problem solution within the company. The evaluation study group did maintain daily contacts with the programmers to assess their opinions about the languages and to keep track of progress so this could be factored into the analysis.

Each programmer's time was recorded for four programming phases: the time devoted to learning and review of the language; the time for design of the program (this included understanding the programming specification, redefining it in the light of the language to be used, and constructing any flow charts necessary to program the problem efficiently); the time for coding the problem (the time starting when the programmer actually began to write code and ending when he was able to make his first debugging run); and last, the amount of time required for checkout. The recorded times were not strictly calendar or clock time, but the amount of time that the programmer was actually devoting to the problem. Nonproductive time such as that spent waiting between computer runs or for keypunching was not counted. An accurate record was kept of time lost because of problems with the compiler or the operating system so that their effects could be factored out.

Quantitative data collected about the runs submitted by the programmers were recorded on daily run logs. These data included the compilation and execution times for each run, the number of statements of each type in the

source program, and descriptive comments about the results of the run. Error logs were also maintained; when an error was discovered during debugging, this fact was recorded and the error was classified according to its type.

After completing the first implementation of a benchmark problem, the programmer was given a qualitative language evaluation form which required 24 essay responses. The following are typical of the kind of subjective questions and statements this form contained:

> "What elements or features of this language seemed to be especially suited for this problem?"

> "Discuss any specific and strict rules which you think tend to cause the programmer to make errors in writing statements."

After completing his responses to this form with reference to language which he had just finished using, the programmer handed them in to the evaluation study group and then started work on the same problem, this time in the second language. Again, all the pertinent quantitative data were gathered during the course of the second implementation. After completion of this second implementation, the programmer answered the qualitative evaluation form for the second language. Since his answers for the first language were no longer available to him as a reference, this was an independent evaluation.

Several days after finishing his second independent set of evaluation essays, the programmer was requested to complete a qualitative language comparison form. This questionnaire, like the first, contained questions applying specifically to the problem, for example:

> "Overall, which language is more suitable for this problem?"

and more general questions:

> "Which language is more suitable for large problems in this application area?"

and questions directed to specific language features:

> "Which language is easier to use in coding of complicated logic control?"

SUMMARY OF LOGICON'S COMPARATIVE EVALUATION OF PL/I

The essential difference between this questionnaire and the first was that this one solicited a direct comparison of the two languages and required that a choice be made wherever possible. The programmer was requested to state a brief reason for his choice.

The complete set of results for each benchmark problem, including detailed tabulations of the quantitative data and all qualitative responses made by the programmers and the evaluation study group, are presented in ESD-TR-68-150. The remainder of this paper summarizes these results.

Quantitative Results

One of the more readily measurable language-dependent aspects is the size of each source program written for problem solution. This is shown in Figure 1, where each bar indicates the distribution of source program statements between executable and nonexecutable statement categories. With reference to the total number of statements, ALOREP2 is the smallest of the PL/I programs, with 214 statements; TSME, which has 703 statements, is the largest. The smallest program written in one of the comparison languages is VIG in FORTRAN, with 295 statements; the largest is the MMI program in FORTRAN, with 1032 statements. The PL/I source program has fewer statements than the comparison language program in all cases except for VIG, a small scientific problem. Most of the difference in size between the PL/I and FORTRAN versions of the VIG problem is due to the fact that there are fewer comments in the FORTRAN program.

Figure 1 indicates that for some problems, the PL/I version is smaller chiefly because it has fewer executable statements. For other problems the PL/I version is smaller chiefly because it has fewer nonexecutable statements; and in some cases the PL/I version has about an equal savings in both executable and nonexecutable statements. Therefore it is not possible to generalize from the results as to whether the executable or nonexecutable language elements of PL/I are responsible for smaller program size. Further breakdowns of program size given in the final report show that there is one area in which the PL/I version is consistently smaller, sometimes by ratios of 15 to 1; this is the data, file, and format description area.

Figure 2 shows the programmer time used in coding and debugging the source programs. Five of the seven problems were coded more rapidly in PL/I than in the comparison language; the two exceptions are VIG, which also was the exception with regard to program size, and SPP-B. However, the debugging time actually spent shows quite the opposite: the PL/I programs

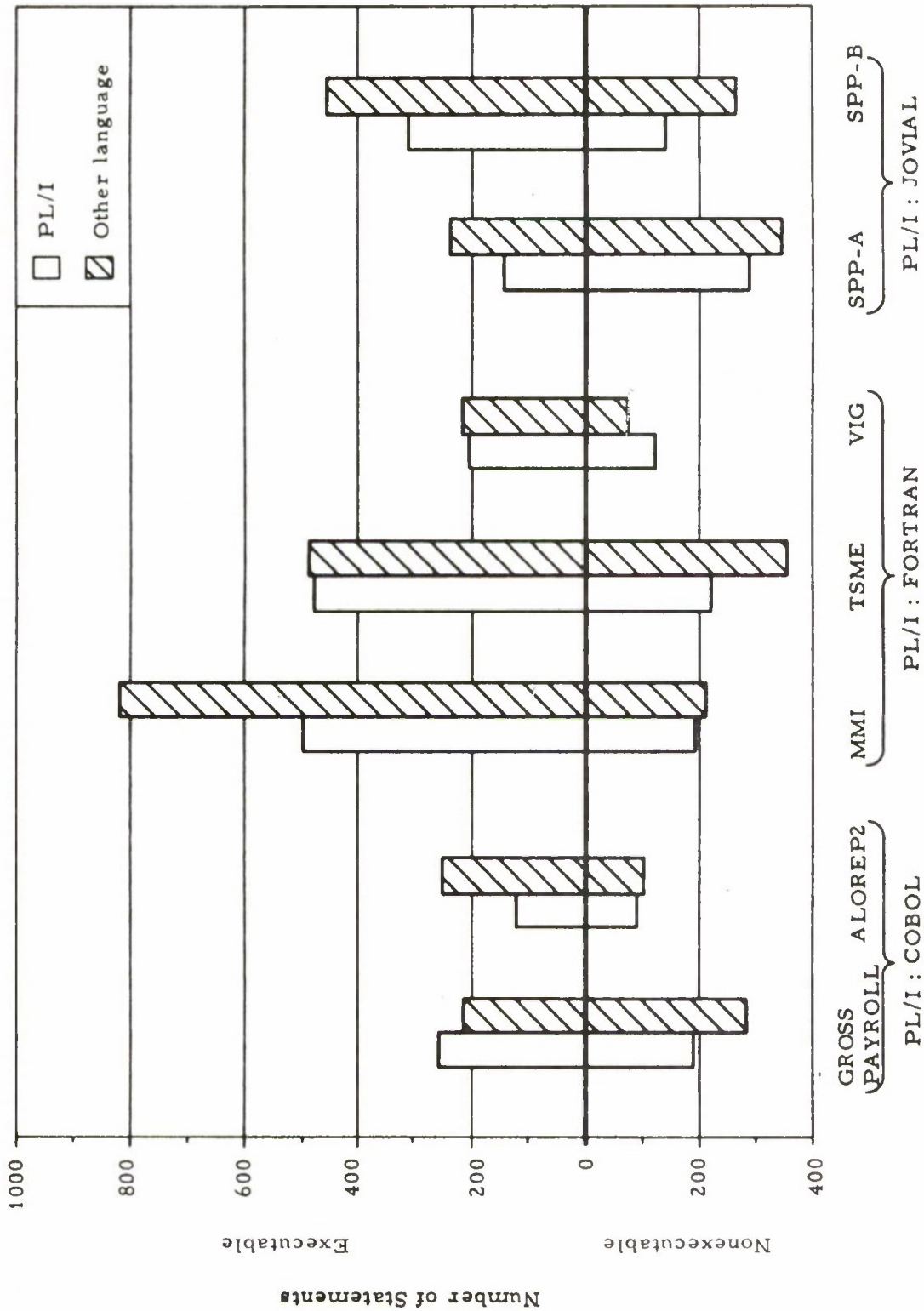


Figure 1. Source Program Size

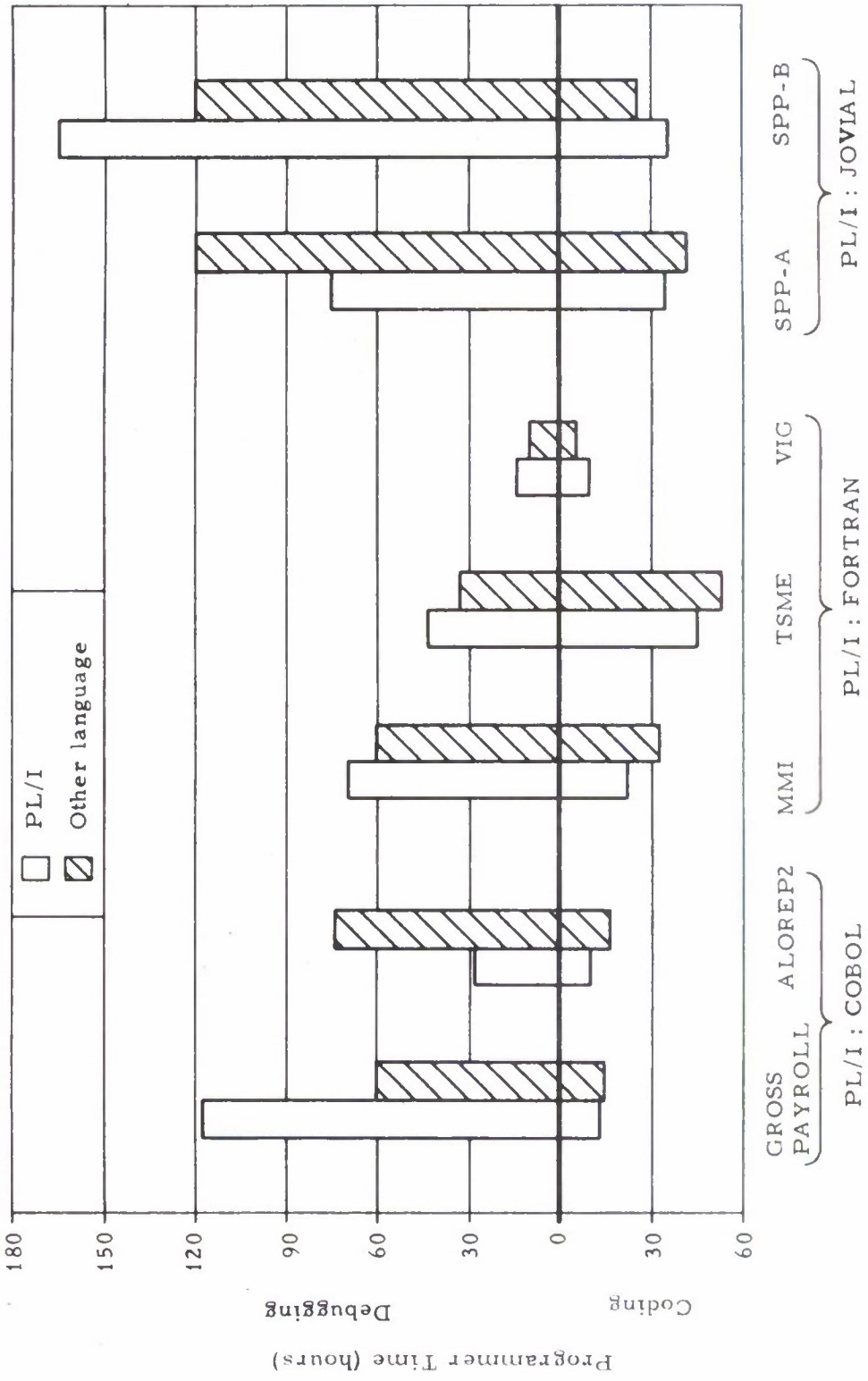


Figure 2. Programmer Time Expenditures

generally took longer to debug even though they were smaller in size than the comparison language versions. The two exceptions here are ALOREP2, for which the PL/I program required a substantially shorter time for debugging, and SPP-A, where again there was a rather substantial difference in favor of PL/I. For the MMI and TSME problems, PL/I and FORTRAN took about the same total time for problem solution, the longer debugging times being offset by the shorter coding times.

Figure 3 combines the data from the first two figures and introduces another factor important to this study: the order of implementation in the two languages. The vertical scale represents coding time in hours and the horizontal scale the relative program size, the first implementation of each problem being considered as having a size of 100%. The MMI problem will be used to illustrate the way in which this graph was constructed. MMI was implemented first in PL/I and took 22 hours to code; hence a point is plotted at 100% and 22 hours. The FORTRAN implementation of this problem took 33 hours to code and resulted in a source program size of 149% relative to the size of the first program. A point representing these values is plotted, and the two points are connected to associate the pair.

On this figure, one language is indicated as superior to the other with which it was compared when the benchmark program written in it is represented by a point lower (less coding time) and to the left (a smaller program) of the point representing the other program. It can be seen that PL/I was the superior language for five of the seven problems. The exceptions are SPP-B, for which the PL/I program was smaller but took longer to code, and VIG, for which the PL/I version took longer to code and was also the larger source program. It is noted, however, that for both VIG and GROSS PAYROLL the differences are small and in the opposite direction. In general, PL/I is indicated as superior regardless of the order of implementation by comparing SPP-A and TSME, which were programmed in the comparison language first, with MMI and ALOREP2, which were programmed in PL/I first.

Figure 4 is similar, the vertical scale indicating debugging time rather than coding time. Unlike Figure 3, this graph fails to bring out any definite trends. Overall, Figure 3 tends to indicate that the smaller size of the PL/I source programs resulted in less time being required for coding regardless of the programmer's previous knowledge of the problem, while Figure 4 suggests that debugging time is not directly related to program size or previous knowledge of the problem but is more a function of experience with the languages, that is, the number of errors made in coding.

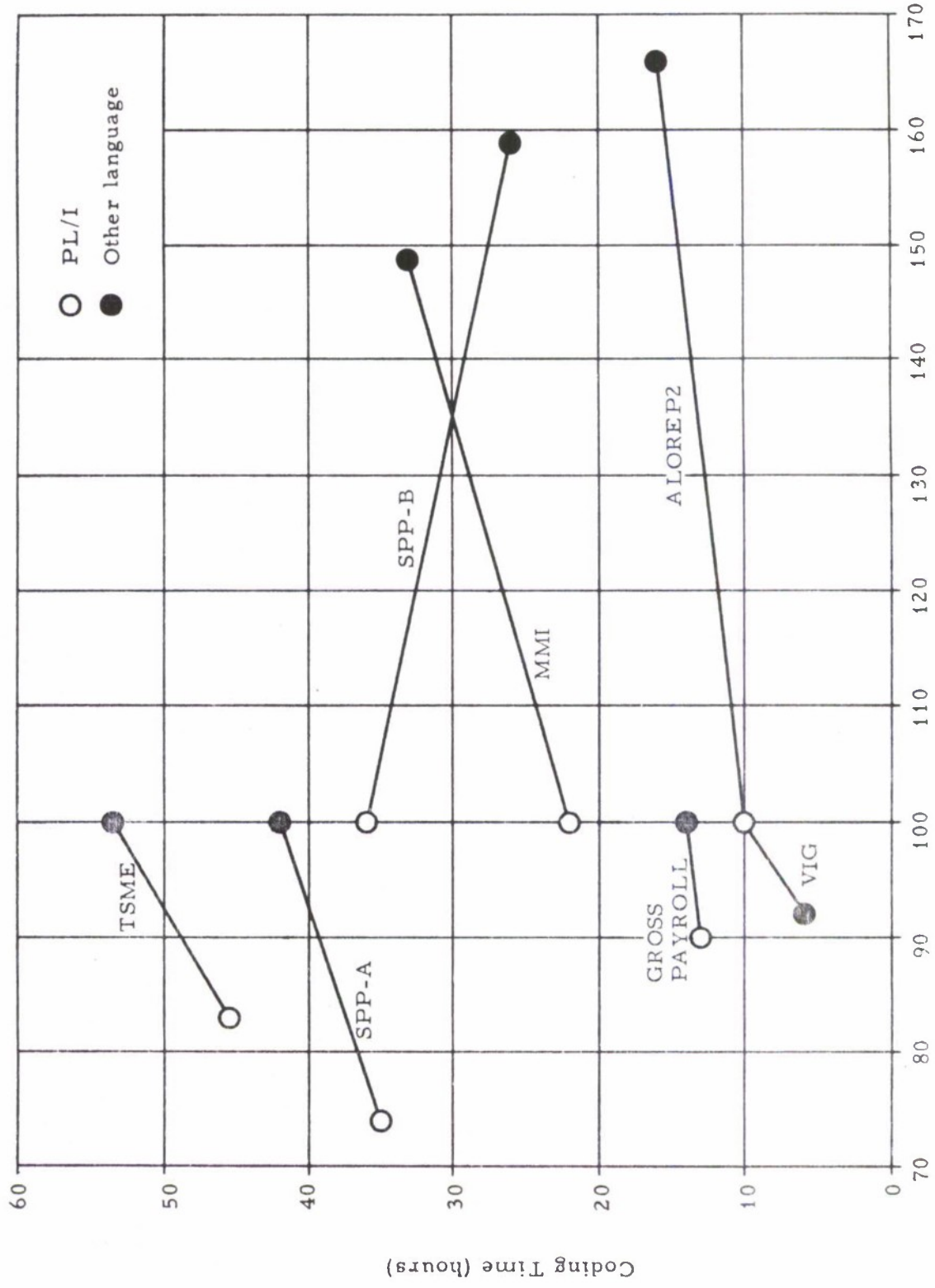


Figure 3. Coding Time vs. Program Size

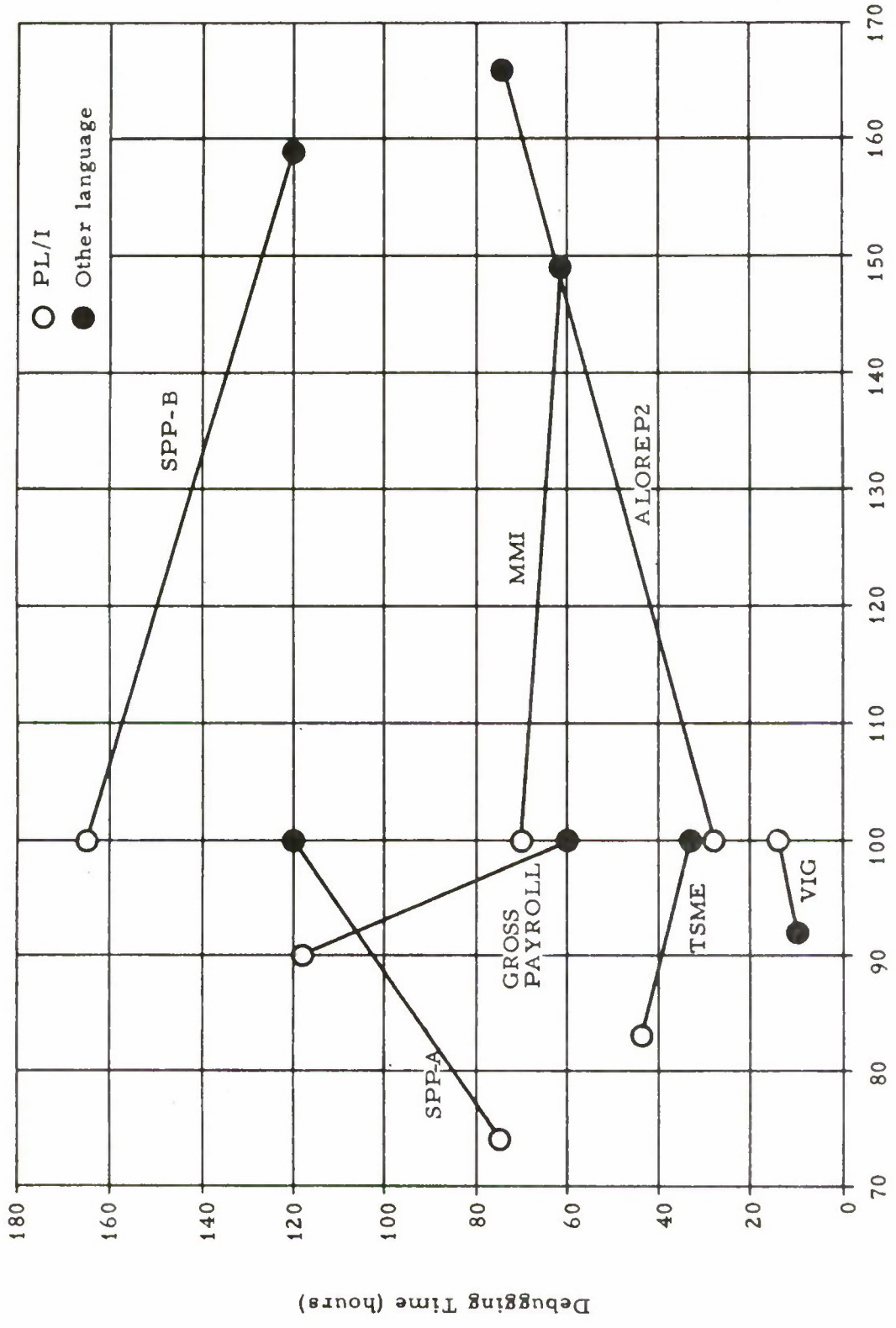


Figure 4. Debugging Time vs. Program Size

Figure 5 shows that there were indeed more coding errors made in PL/I, even though the PL/I versions were generally smaller. In this chart the clear bars show the sums by error category for all PL/I programs, the shaded bars the sums by error category for all comparison languages combined: COBOL, FORTRAN, and JOVIAL. If comparable figures were drawn for each individual problem, their shapes would not be substantially different from that of the composite given in Figure 5.

In all but two categories there were more errors made in the PL/I versions; the exceptions are computation and assignment statements and labels. The error most frequently made was the omission of the semicolon from the end of PL/I statements. The large difference in the punctuation category is at least partly due to the fact that the programmers were not as familiar with the PL/I punctuation rules as with those of the comparison languages, and it is expected that this and other punctuation errors will decrease in frequency as programmers gain more experience in using PL/I. The same is probably generally true of the time required for debugging.

For both PL/I and the comparison languages, many errors were made in the data, file, and format description category. The number in this category is far greater than in the sequence control and decision and the computation and assignment categories where the majority of attention in problem solution and even debugging is often concentrated. As stated earlier, PL/I showed a substantial savings in data, file, and format description statements. It can be expected that as programmers become more experienced in using these superior capabilities of PL/I, which are evidently a little more complex, errors in this category should decrease.

Qualitative Results

Turning now to the subjective, qualitative opinions of the programmers and the evaluation study group, the responses to the qualitative language comparison form are summarized in Tables 2 through 5. In these tables the top row of dots for each entry represents the choice of the programmers, the bottom row the choice of the study group. These tables are to some extent oversimplifications, since it has been impossible to include on them the reasons given for each choice. In cases where there is a conflict between a programmer's choice and that of the study group, it is important to recall that the selections made by the group were based on their analysis of the responses to all questionnaires, on the numerical data, on their knowledge

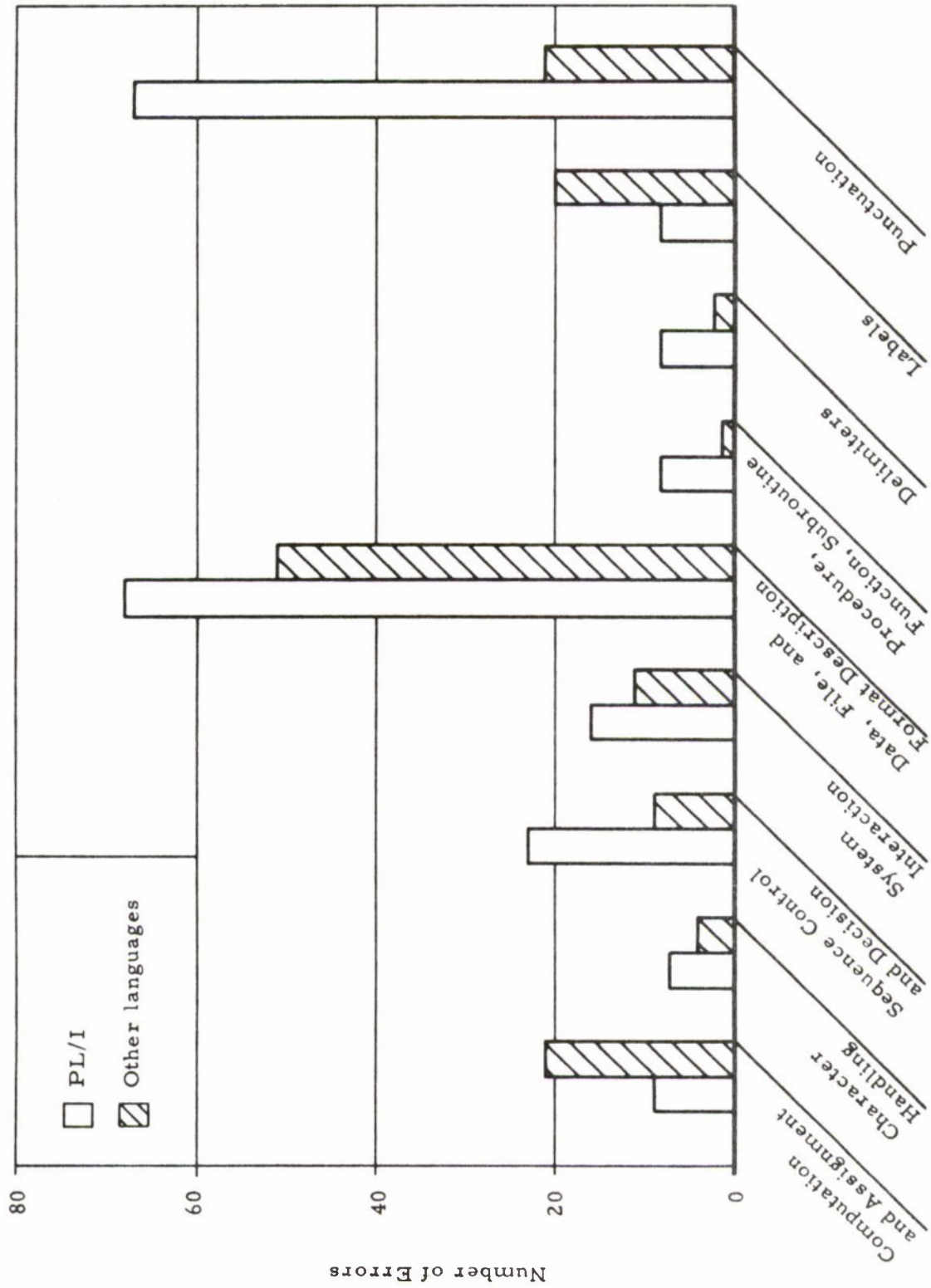


Figure 5. Error Sources by Category for All Programs

about the programmer, and on their daily contacts with him. The complete set of replies to all questionnaires appears in ESD-TR-68-150.

The first qualitative opinion recorded in Table 2 refers to which language is more suitable to the particular benchmark problem. PL/I is preferred for all benchmark problems except GROSS PAYROLL, the dollars-and-cents business problem, and VIG, the small scientific problem. When the suitability of the languages for small and medium problems in the application area is considered, PL/I is the choice except in the business application area, COBOL being chosen for both dollars-and-cents and business data management problems. For large problems, PL/I is again chosen as more suitable than FORTRAN or JOVIAL, while the consensus is that there is no preference favoring either PL/I or COBOL for the solution of large business problems.

The next characteristic examined in Table 2 concerns suitability for the nonprofessional programmer, which for the purpose of this evaluation was

Table 2. Language Suitability

Key: O PL/I ● Other language -- No preference	COBOL		FORTRAN			JOVIAL	
	GROSS PAYROLL	ALOREP2	MMI	TSME	VIG	SPP-A	SPP-B
For this problem	●	O	O	O	--	O	O
	●	O	O	O	--	O	O
For small/medium problems	●	●	O	O	--	O	O
	●	●	O	O	O	O	O
For large problems	O	--	O	O	●	O	O
	--	--	O	O	O	O	O
For nonprofessional programmer	●	O	●	●	●	O	O
	●	●	●	●	●	O	O
For professional programmer	O	O	O	O	O	O	O
	O	O	O	O	O	O	O

defined as a person who is interested not so much in the details of programming as in solving a problem to obtain the answers he needs for his primary field of interest. Such people normally program infrequently, and then only in a restricted application area. All of the programmers and analysts participating in the evaluation are professional programmers, and all of them have extensive experience assisting nonprofessionals in the solution of problems; that is, they all are familiar with the ways in which nonprofessionals have gone about solving programming problems. The choices of these professionals with regard to suitability of the languages for the nonprofessional favor PL/I only in comparison with JOVIAL for the data management application area. There is almost complete agreement that COBOL is better suited for the nonprofessional in the business area and FORTRAN for the nonprofessional in the scientific, interactive, and simulation and gaming areas. For the professional programmer, however, PL/I is the unanimous choice of all participants.

Table 3 shows one more level of detail by bringing out some of the gross language characteristics that help to determine a language's suitability. With regard to generality, PL/I is the unanimous choice except for business

Table 3. Language Characteristics

Key: ○ PL/I, ● Other language, -- No preference	COBOL		FORTRAN			JOVIAL	
	GROSS PAYROLL	ALOREP2	MMI	TSME	VIG	SPP-A	SPP-B
Generality	○	--	○	○	○	○	○
	--	--	○	○	○	○	○
Conciseness	○	○	○	○	●	○	○
	○	○	○	○	○	○	○
Naturalness	●	--	○	○	○	○	○
	●	--	○	○	○	○	○
Conciseness : naturalness	●	○	○	○	○	○	○
	○	○	○	○	○	○	○

programming. Even though PL/I is usually considered a more general-purpose language than COBOL, there is a consensus that PL/I has serious deficiencies in the business programming area, particularly its lack of built-in sorting and report writing capabilities; thus no choice could be made between the two languages. For conciseness the choice of PL/I is unanimous except for the programmer of the VIG problem. For naturalness to the particular benchmark problem, PL/I is chosen over JOVIAL and FORTRAN but not over COBOL; in fact COBOL is considered more natural than PL/I for the dollars-and-cents type of business problem.

The last set of qualitative choices shown on Table 3 concerns the balance between conciseness and naturalness. A language in which one could code a very concise program with the absolute minimum number of statements would almost certainly not be a very natural language to use. On the other hand, the most natural language to use, presumably English, is not nearly concise enough. PL/I is chosen almost unanimously as providing the best blend between the two characteristics.

Table 4 presents the answers to some questions regarding programmer reactions to practical language aspects. Almost all responses show the comparison language to be easier to learn, whether COBOL, FORTRAN, or JOVIAL. For ease of using the language, the choice is generally PL/I. The explanations given in the essay answers clarify what might be considered a contradiction between these two conclusions. FORTRAN, for example, being more limited than PL/I, is more easily learned. But when it comes to solving a specific problem, the more extensive capabilities of PL/I make it necessary to employ few if any tricks to accomplish the task, while the FORTRAN programmer may have to resort to many programming tricks and in some cases assembly language to do the same job.

With regard to ease of learning and using for the particular problem, PL/I is chosen for five of the seven problems. The responses concerning the ease of reading and writing statements show everyone choosing PL/I except in comparison with COBOL. For productivity in coding the benchmark problem, which refers to the amount of time required after receiving the specification until the program has been designed and coded, PL/I is generally chosen over FORTRAN; there is a division for the two business problems; and no preference is indicated between PL/I and JOVIAL for the two SPP problems. Note that the programmers' subjective opinions agree quite well with the amount of time they actually spent.

SUMMARY OF LOGICON'S COMPARATIVE EVALUATION OF PL/I

Table 4. Programmer-Dependent Aspects

Key: ○ PL/I, ● Other language, -- No preference	COBOL		FORTRAN			JOVIAL	
	GROSS PAYROLL	ALOREP2	MMI	TSME	VIG	SPP-A	SPP-B
Learning	●	○	●	●	●	●	●
	●	●	●	●	●	●	●
Using	●	○	○	○	●	●	○
	--	--	○	○	○	○	○
Learning and using for this problem	●	○	○	○	●	○	●
	●	○	○	○	●	○	○
Reading and writing	●	--	○	○	○	○	--
	●	●	○	○	○	○	○
Productivity	●	○	○	○	●	○	●
	●	○	○	○	○	--	--

Table 5. Ease of Use

Key: ○ PL/I, ● Other language, -- No preference	COBOL		FORTRAN			JOVIAL	
	GROSS PAYROLL	ALOREP2	MMI	TSME	VIG	SPP-A	SPP-B
Logic control	○	○	○	○	○	--	○
	○	○	○	○	○	--	--
Unit manipulation	●	○	○	○	○	●	○
	○	○	○	○	○	--	--
Statement modification	○	○	○	○	○	○	○
	○	○	○	○	○	○	○

Table 5 delves further into several language aspects that enter into ease of use. With regard to capabilities for logic control, PL/I is chosen except when compared with JOVIAL. For unit manipulation (by which is meant bit, character, and byte manipulation) PL/I is again preferred except in comparison with JOVIAL. Finally, for ease of statement modification, which refers to the ease with which a statement can be corrected when an error in the program is found, PL/I is the unanimous choice.

Compiler- and System-Dependent Data

Although the emphasis of the study was on the languages themselves, a great deal of information was perforce generated about the performance of the computer systems and compilers utilized; these are identified in Table 6.

Figure 6 shows the computer time required to run the checked-out programs, with compilation time at the top and loading (link-editing time for IBM 360 versions) and execution time for a typical test case at the bottom. For all

Table 6. Computer Systems and Associated Compilers

Language	Benchmark Problems	Computer System	Compiler
PL/I	All	IBM 360/65 ASP	Level F Version 2
COBOL	GROSS PAYROLL	IBM 360/65 ASP	Level F
	ALOREP2	IBM 360/65 ASP	Level F
FORTRAN	MMI	IBM 360/65 ASP	Level H
	TSME	IBM 360/65 ASP	Level H
	VIG*	UNIVAC 1108	FORTRAN IV
JOVIAL	SPP-A	GE 635	Version 36
	SPP-B	GE 635	Version 36

*The VIG FORTRAN program was also compiled and run on the GE 635, and compilation time was obtained on the IBM 360/65 ASP system using the Level H FORTRAN compiler.

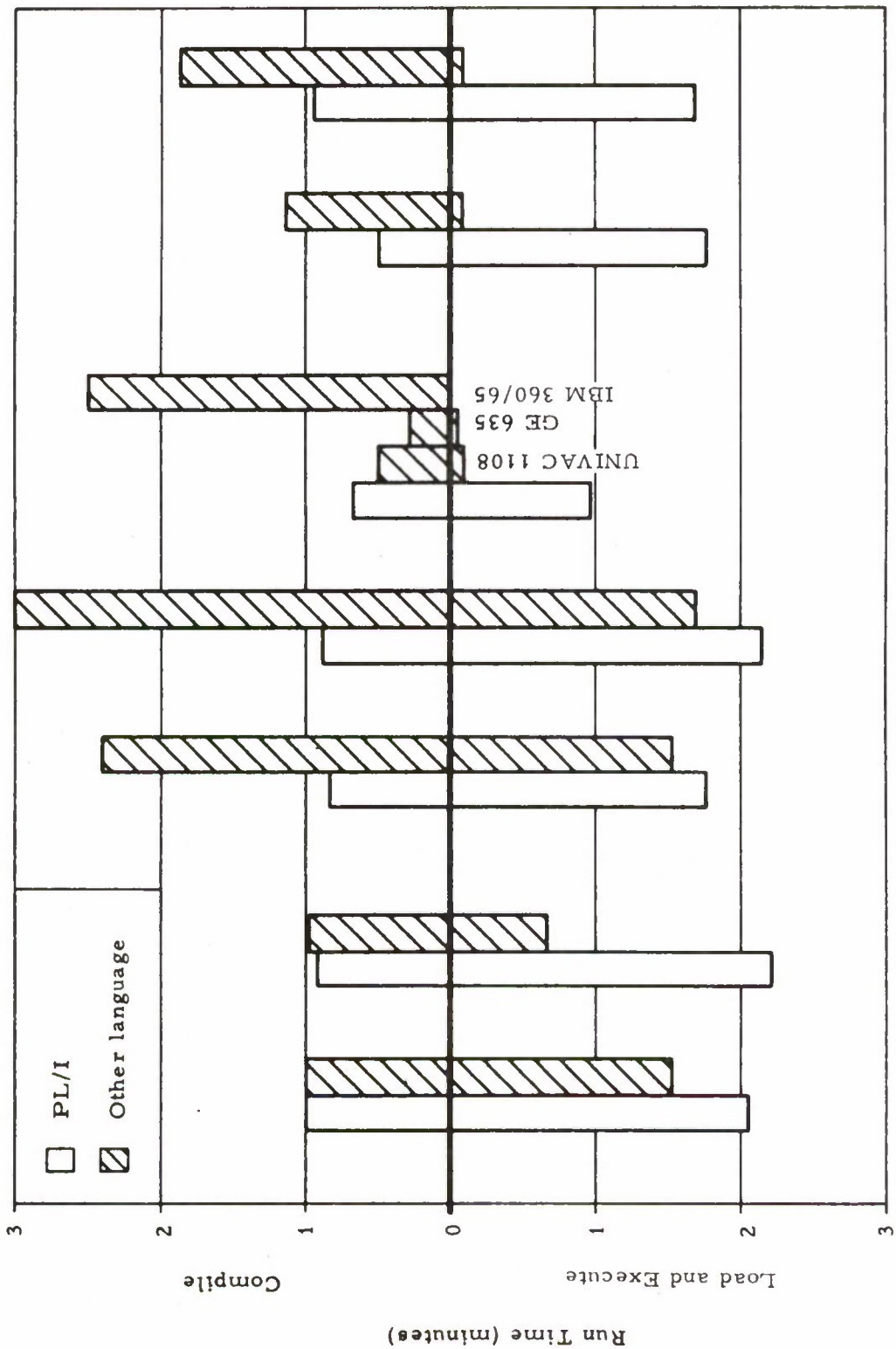


Figure 6. Program Run Time

problems except VIG, the PL/I compile times were shorter than or equal to those for the comparison language versions, and for all problems the PL/I link-editing and execution times were longer than the loading and execution times obtained for the comparison language programs. For VIG, the compile time for the UNIVAC 1108 FORTRAN version is somewhat less than the IBM 360 PL/I compile time, the 1108 FORTRAN load and execute time is but a tenth that of the PL/I version, and the GE 635 FORTRAN times are even shorter than the UNIVAC 1108 times. While the IBM 360 Level H FORTRAN compile time is considerably longer than that obtained for the other two FORTRAN compilers, it seems to be compatible with the FORTRAN compile times obtained for the MMI and TSME programs. No link-edit and execute time could be obtained on the IBM 360 for the VIG FORTRAN program; while this program was acceptable to the UNIVAC 1108 and GE 635, several compile errors were encountered when using the IBM 360, these errors involving features present in the other FORTRAN compilers but missing from Level H.

Another interesting type of information that can be gleaned from the data relates to the number of runs required to complete debugging; these are shown in Figure 7. Runs were sorted into two classes: those made to correct syntax errors and those that were unproductive because of difficulties with the system, i.e., required because of a problem in the compiler or the operating system or a bad reaction to otherwise good control cards. Programming errors are defined as those in which a syntactically correct statement is written, but not the right statement to solve the problem. In some cases the number of runs found to be unproductive because of system trouble was nearly a third of the total number of runs. The high number of system-type errors is probably indicative of the state of compiler and operating system development for the newer, more advanced systems.

Total debugging cost for each benchmark problem is shown in Figure 8 in terms of minutes of computer time required. In all but one case, TSME, PL/I required more computer time than the comparison language did for debugging. The relative differences are not too substantial, however, except for the VIG and SPP problems.

Figure 9 shows the sizes of the object code programs. The IBM 360 object programs, which are measured in bytes (8 bits per byte), were divided by 4 so that they could be compared with the UNIVAC 1108 and GE 635 object programs, which are measured in words (36 bits per word). For the two business problems, the PL/I Level F Version 2 compiler generated considerably larger object programs than the Level F COBOL compiler. However, the sizes of the generated object programs are comparable for the

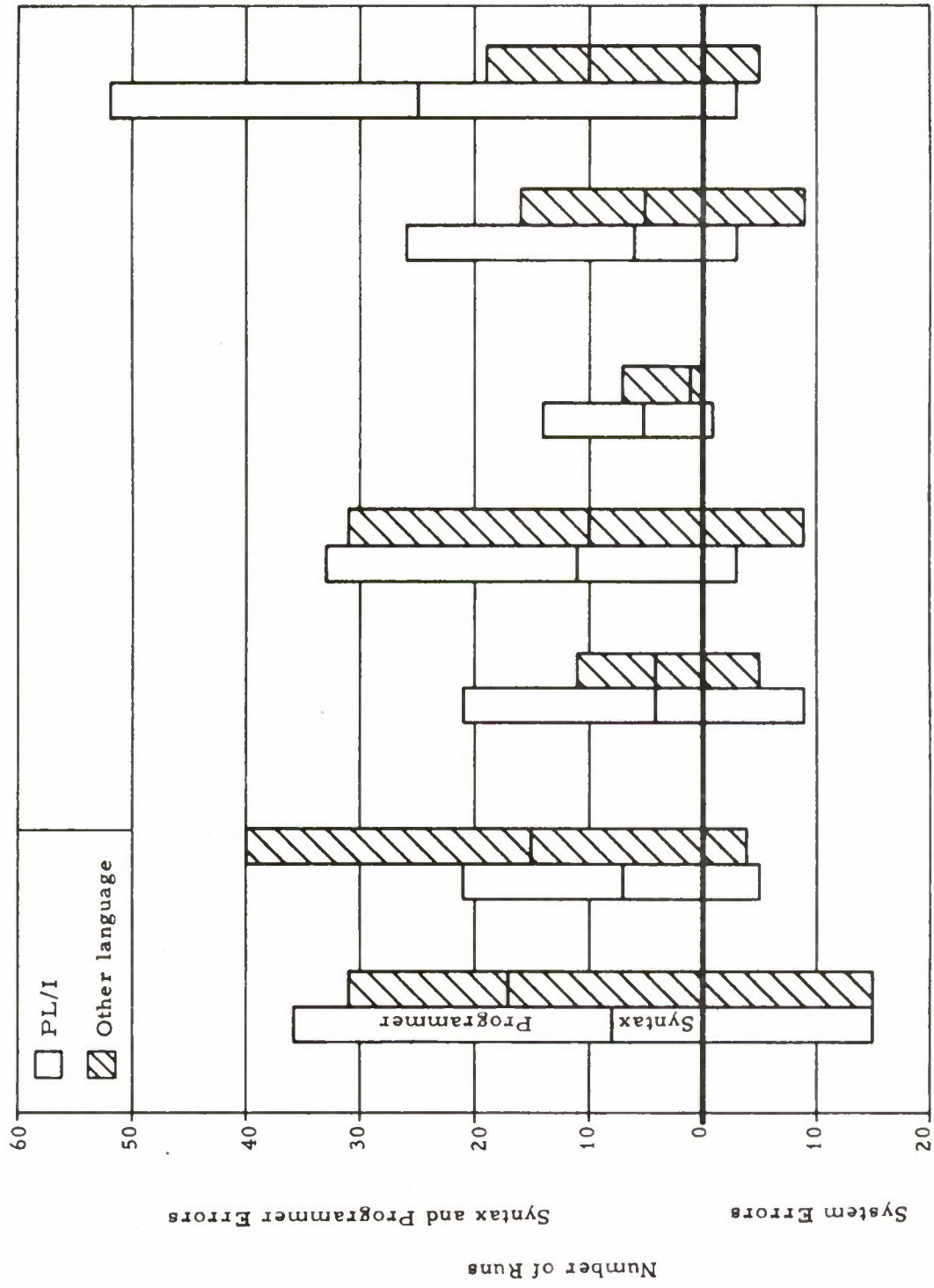


Figure 7. Number of Debugging Runs

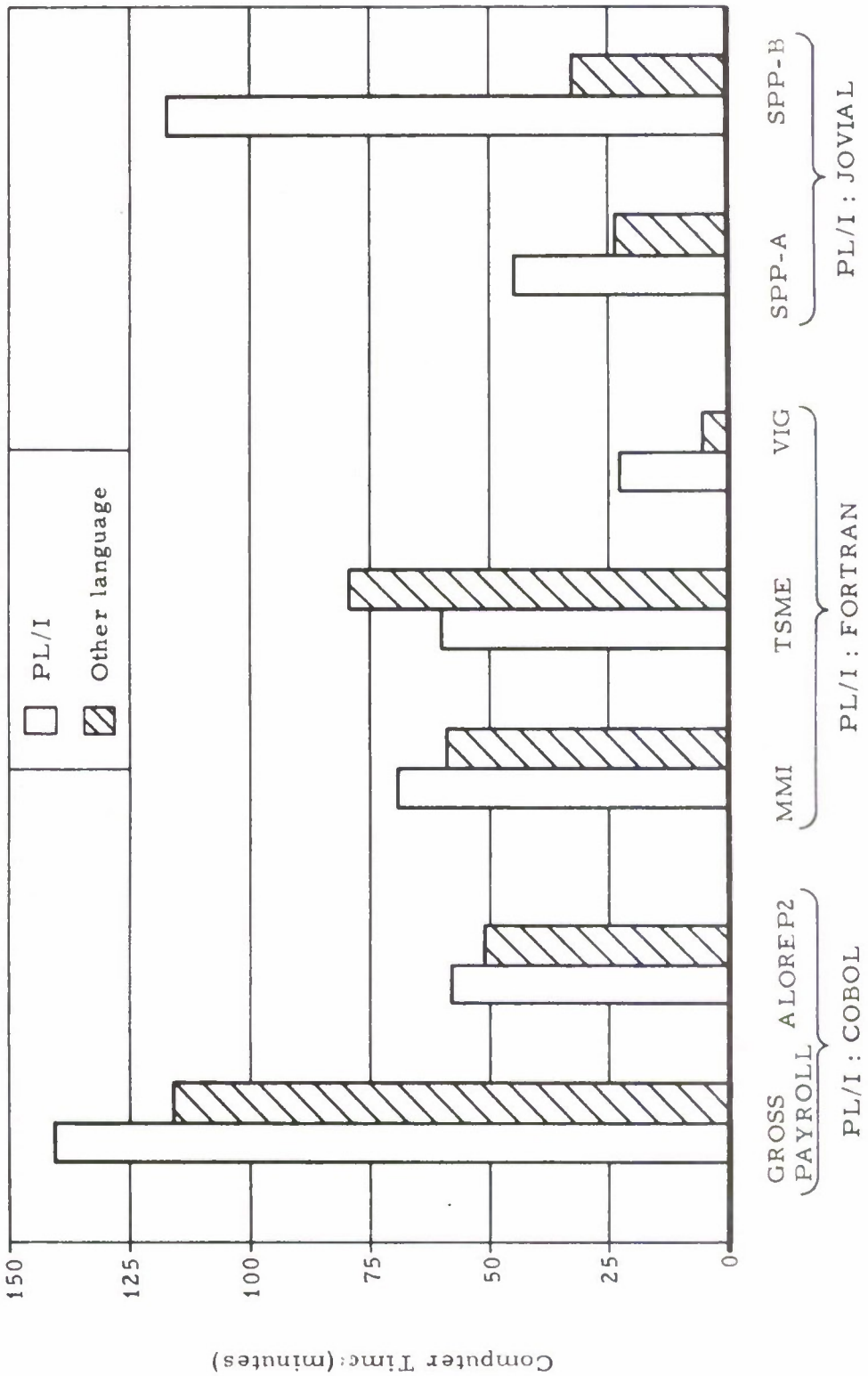


Figure 8. Computer Time Expended

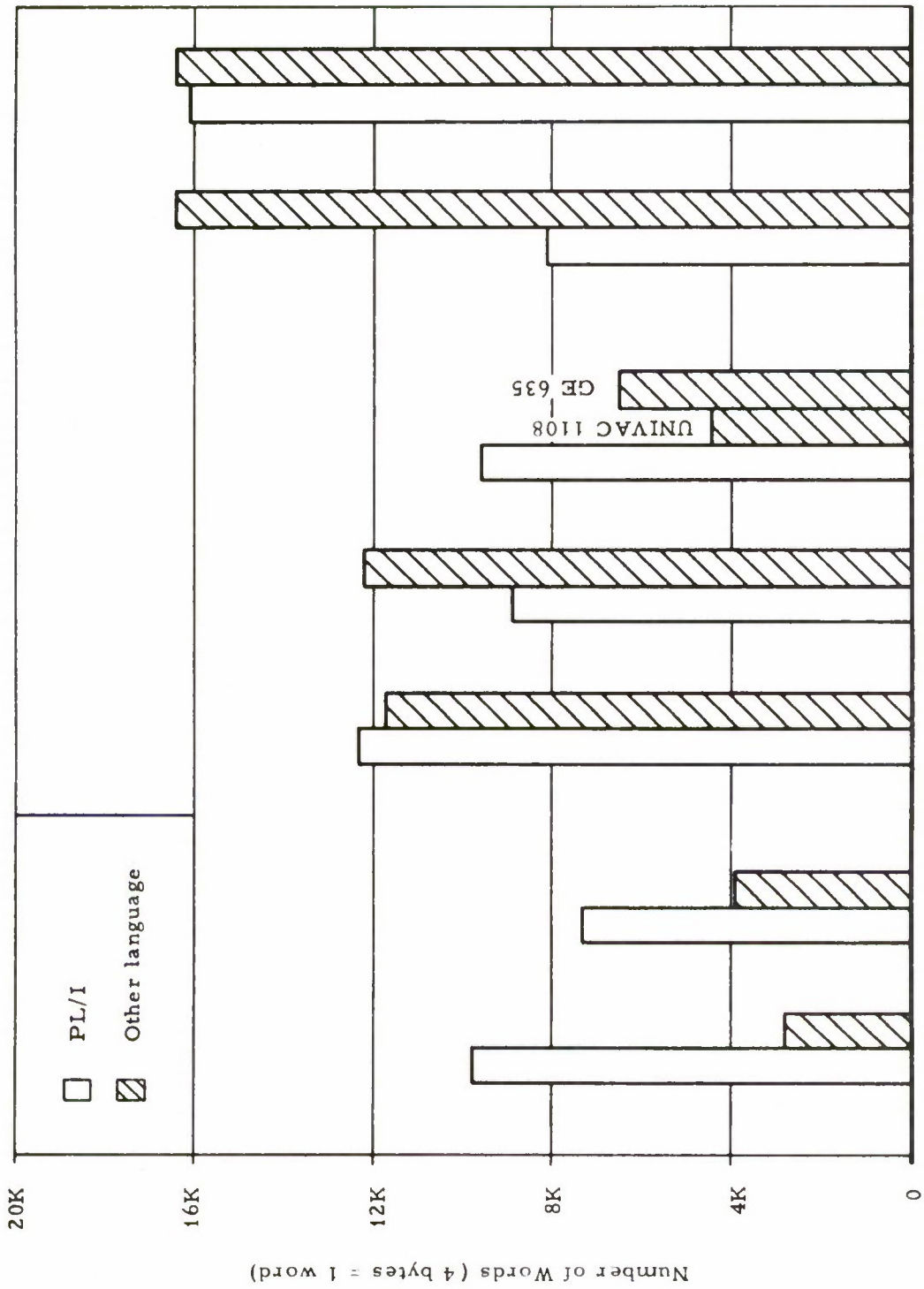


Figure 9. Object Program Size

PL/I and FORTRAN compilers used for MMI and TSME. For VIG the PL/I compiler produced a considerably larger object code program than did the UNIVAC 1108 and GE 635 FORTRAN compilers. The object code program produced by the SPP-B JOVIAL version was the same size as the SPP-B and SPP-A PL/I versions, but the SPP-A JOVIAL version was half that size. One interesting result is that the large differences in sizes of the SPP source programs do not correspond directly with object code program sizes. It is apparent that a significant difference in both source and object code sizes is possible for programs produced by people who have the same experience and are solving the same problem.

Conclusions

It seems evident that there is no order-of-magnitude improvement in changing from one of the comparison languages to PL/I, the kind of improvement one would expect when going from assembly language to FORTRAN, for example.

PL/I is more difficult to learn than any of the comparison languages. However, none of the programmers had particular trouble in this area, so it should not be classed as an extremely difficult language to learn. The extra capabilities of PL/I indicate that it is a good investment for the professional programmer to spend the extra time to learn it.

The PL/I user can probably effect a savings in coding time right away. However, debugging time is longer than that for any of the comparison languages until the programmer becomes experienced in debugging with PL/I.

As shown by VIG, the small scientific problem that had approximately 300 statements and was largely just mathematical equations, PL/I probably complicates simple mathematical problems.

The nonprofessional programmer is probably going to be better off with small, simple, more specialized languages.

The results of the SPP-A and SPP-B implementations indicate that the difference between any two languages is less than the difference between two programmers that, based on their background and experience, would be considered equivalent; the programmer is a greater influence on the problem than is the language.

SUMMARY OF LOGICON'S COMPARATIVE EVALUATION OF PL/I

PL/I generally requires fewer statements to solve a given problem than COBOL, FORTRAN, or JOVIAL. The size reduction is particularly significant in the data, file, and format description area.

Programmers tend at first to make more errors in writing PL/I programs than when using the comparison languages; this is especially true of punctuation errors. Errors in data, file, and format description were the most common errors for all of the languages used in this study. In this area PL/I's better capabilities should eventually favor it relative to the other languages.

Finally, compiler and operating system considerations can completely override other factors in choosing a language. This is particularly important if the user is especially concerned with the time it takes both to compile and to execute programs and the amount of system troubles he is likely to encounter. It is in this area that improvements are most urgently needed, for both PL/I and the comparison languages.

Questions and Answers

Question: Your quantitative data regarding the VIG program seem to indicate a disparagement or a bad situation for PL/I. Your comments also indicate that. However, on the slides you showed detailing the programmers' comments and the committee's comments, it seemed to me that you were recommending PL/I for the most part all the way through as far as the scientific programs went.

Answer: That's true. For the qualitative or subjective data the people felt PL/I was better, even though they couldn't substantiate this in all the cases with numerical or quantitative data to back them up, particularly for the VIG problem. It seems that a bigger scientific problem would have been desirable, something that was more of a test of PL/I. The VIG problem tended to be the same in PL/I and FORTRAN; it didn't call on the expanded capabilities of PL/I. I think that would show why the numerical data would be close, or even show PL/I at a disadvantage for that problem, but the subjective responses would indicate PL/I was better.

Q: Are you planning to run these PL/I problems through Version 3 of the PL/I compiler, and if you already have, can you comment?

A: Right now we have no plans on running them through the Version 3 compiler because, as I said, we were interested more in the language aspects. Unfortunately, we had to run through existing compilers. What we would have liked to have done, and we originally planned to do, was to use another manufacturer's compiler to get an idea of the system troubles to be expected. Maybe that would have been better, but we were limited to one PL/I compiler. That's one of the reasons we stayed with the Level H FORTRAN compiler. We didn't want to switch over and use a FORTRAN that we knew might be better, and have less system troubles, because we felt we would be putting PL/I at a handicap.

Q: Did you consider using the DOS subset PL/I? Why did you pick FORTRAN H over FORTRAN G in the 360?

A: We used what was readily available to us at the service bureaus and computer centers.

Q: Would you care to comment on what the programming experience was? You've given the number of years of programming experience for your people. Would you care to comment on what their previous knowledge of programming languages was? How many years?

A: The ALOREP2 programmer had about one month COBOL, one month PL/I, and the rest either FORTRAN or machine language in that one year. The GROSS PAYROLL programmer had most of his experience with Autocoder. I don't think he had a great deal of experience in either PL/I or COBOL. The TSME programmer had about two years of FORTRAN experience and the rest machine language. The MMI programmer had about a year of PL/I experience, including some PL/I systems work, and several years of FORTRAN experience. The VIG programmer had a couple of years of FORTRAN, and her previous PL/I experience was limited to a two-week IBM training course in PL/I. Neither one of the SPP programmers had any prior JOVIAL or PL/I experience.

Q: How did you get your PL/I experience, your knowledge? What kind of education?

A: You mean the individual programmers? That was up to the individual programmers. They were given manuals and texts. We didn't conduct any formal training classes. There was a lot of interchange between the programmers as the project developed, but we didn't specifically try to set up a rigid plan for educating them in PL/I.

Q: Just by way of information, the COBOL F is really COBOL G. I think it's important to try to find out which version of FORTRAN H you used, whether it was the optimizing or the zero level. Was the JOVIAL programming all in JOVIAL or did they resort to machine code?

A: That's a good point. We did a considerable amount of research in JOVIAL trying to get a JOVIAL compiler that was for the whole JOVIAL language. We had to settle for doing the input/output portions in machine code. We couldn't find any available JOVIAL compiler which handled the input/output very well.

Q: That might have influenced a great deal of the quantitative results that you got. You were running on a GE?

A: GE 635. If you remember, we had a considerable number of system problems there. Again, we used a JOVIAL compiler that's sort of in the development stage, too, maybe like PL/I. It's not by any means a finished compiler.

Q: The number of runs you made seemed to be rather higher than I would expect. Were you intentionally trying to use machine time at the expense of programmer time?

A: No; we weren't trying to intentionally bias it either way. This was what the programmers felt they needed. We do think these numbers are consistent with the number of runs required to develop other comparable programs. A lot of the runs were, again, due to system problems, however. In fact, the number of runs probably is partly reduced from what would be expected in an installation where the programmer gets very rapid turnaround. I think that most of the programmers were getting two runs a day at the very best.

Q: In your observations on FORTRAN, comparing it with PL/I, the people overwhelmingly thought it was easier to learn FORTRAN but harder to use it. I thought your explanation for this had to do with the tricky things that you had to do in FORTRAN that you didn't have to do in PL/I. Were the problems that you were doing here such that they required the programmers to use these tricky things in the FORTRAN work they did?

A: I don't think that they were required to use any special new tricks. There were tricky things that they already knew and probably used naturally, but they just had the feeling that for many problems they would have to use tricks.

Q: With COBOL it is not easy to divide the program up into subroutines, but FORTRAN does have this feature. Did you use subroutines, subprograms, to any great extent?

A: No, not in the COBOL problems. In fact in one of the essay answers, the programmer down-rated COBOL for large problems because he didn't feel it was natural to use subrotines in COBOL. I don't think in here that they used subroutines to any great extent for any of the various problems.

Q: None of your programs got into the area of overlay or chaining?

A: No, none of them required overlay or chaining.

Q: I am making the assumption that you did not have to support any of your COBOL or FORTRAN work with machine code. Is that true?

A: That's true, nor was machine code required for the PL/I versions, in fact. All the debugging was done at the source code level. Nobody spent time looking at machine code. Everything except the I/O for the JOVIAL versions was done at the source code level.

Comment: I'd like to make a comment about the semicolons. That is a problem in PL/I when you're starting to learn it. COBOL and FORTRAN and traditionally, I think, most high-level languages in the last 10 years are oriented to the card image, and they use that as a statement delimiter, and it's sometimes a little difficult

to get used to putting the semicolons on. But I think after a good deal of experience, it becomes rather automatic. To get around this problem, in FORTRAN and COBOL specifically, they have specially designed coding sheets.

The tape ended at this point, shortly before the conclusion of the question-and-answer session. A large part of the panel discussion which followed dealt with this presentation.

SECOND PANEL DISCUSSION

Participants

Earl O. Althoff, Eastman Kodak Company
Fernando J. Corbató, Massachusetts Institute of Technology
Vilas D. Henderson (Moderator), Logicon
Mary D. Lasky, The Johns Hopkins University
Robert F. Rosin, Yale University
Raymond J. Rubey, Logicon
J. Michael Sykes, Imperial Chemical Industries, Ltd.

Henderson: Since there was some rousing discussion about our presentation, I would like to start by giving each of the panelists a chance to comment on it before we go into more general questions. Let's start with Bob Rosin.

Rosin: I've got several points that I'd like to make. I'd like to preface my remarks by saying I really don't have a bone to pick with the presentation, not a single bone. I don't think it's overall bad; as a matter of fact, I think it's good. I would say that trying to do the kind of thing that you did in those experiments is one of the most difficult things to do in all of the world of programming right now, and that's why reports on this kind of work are so seldom found in the literature, the journals. I also don't—believe it or not—have a vested interest in any one language or even in any one family of languages. It may not always sound that way from things I say here or at SHARE, but it's true. If I believed in any language at all right now it would have to be PL/II or ALGOL 68 or one of their derivatives—believed in the sense of what I really should be working in five or ten years from now. In terms, however, of a practical tool for the current generation, for people to be using now and young people to be learning now, frankly I feel that PL/I is, in its class, superior. I also think, for example, that SNOBOL4 in its class is superior. That's another issue. Anyway, I wanted to say that to make my remarks sound a little more from a different position other than one close in.

I hope that Vi's introductory comments to the presentation this afternoon will be made not only at the beginning but also at the end of whatever further presentation may be made to the Air Force so that it's well understood that this was one case study

and one limited environment; and that goes for both positive and negative aspects of the presentation.

It's my impression that the people who were involved in the study, the people who were used as the guinea pigs, never learned PL/I. It may be that they learned FORTRAN and COBOL and/or JOVIAL. But it's quite clear to me that they never learned PL/I because frankly I didn't learn PL/I my first program and I don't know of anybody who did, including the present manager of the SHARE PL/I project— who, as was pointed out, was once manager of the SHARE FORTRAN project and has many years of experience. I am quite puzzled why, when PL/I was introduced — or, as a matter of fact, when any of the languages were introduced— no hints or guidance were given to your programmers on use of the language. I'm also curious as to what documentation was provided for learning PL/I, because it's clearly understood that manual C28-6571 dash whatever is not a tutorial document. C28-8201 may be, in some peoples' minds, but I doubt that it was even available. I am curious what you availed yourselves of. I would find the whole prospect frightening, frankly. I am reminded of something I learned when I was a graduate student in psychology, for the one year that I could take it. One of my instructors demonstrated to us that there was no such thing as one-trial learning, that the one trial itself was really the zeroth trial, and what you measured at the end of that first trial was not really learning at all but just the summation of that one trial. I'd be very curious to see what would happen as a follow-up to this kind of study if identically the same people were, say, switched onto different problems; that is, if the programmer who did the payroll problem this time did the other business problem and vice versa—I'd be very curious to see what sorts of things happened, both in COBOL and in PL/I.

Again, I think this kind of study is relatively pioneering. I would say from a statistical point of view that the evidence is rather small in all of your work. As Corby said earlier today in regard to another matter, there is some variance there. It would be interesting to discover what it was.

In looking at the number of statements that were used to code any particular problem, I find it unbelievable that you don't have in the presentation that you made— and I assume in the final report — a breakdown not only between executable and nonexecutable statements but also of comments. The uncommented FORTRAN program to me sounds incredible. On the other hand, it is so easy to sit down at a keypunch and bop in a /* and go on for six pages of PL/I comment that I don't know exactly how you would measure that. It's also not clear to me when you talked about comments whether a single /*...*/ in PL/I was a single comment even if it went on for a page and a half or two pages; I don't know. If you counted a comment card in FORTRAN and then counted one of those occurrences in PL/I as a single comment, as a single statement, then PL/I comes out way, way ahead, and probably in fair measure.

Several people, when we were sitting out in the audience, suggested that a more reasonable experiment, at least to even out the data with respect to PL/I — again, either positively or negatively — would be to get some programmers with zero years of experience. Mary has taught some of these. We have some people at various installations around the country who have learned PL/I as their first language rather than second or third, and these people come through pretty well. They don't come with built-in prejudices such as: "What is this semicolon?" or "Why did I leave it off?" et cetera. They come through not knowing that what they're learning is supposed to be more difficult than what went on in the past. This is quite comparable to my own teaching experience when I taught FORTRAN in the first semester of a replication of the same course, and in the second semester I taught MAD, which is an ALGOL 58 dialect not unlike some aspects of JOVIAL. I found the transition not too terribly difficult, but it was very interesting to watch people who had first learned FORTRAN and then learned MAD and, vice versa, people who had first learned MAD and then learned FORTRAN. The effects were traumatic.

I say again, as was pointed out from the audience, that in your slides on load and execution time, you really ought to break

out the link-edit time. It's just unfair to saddle PL/I or FORTRAN or COBOL with the link-edit time that accrues from OS 360; it's terribly unfair to any of them.

I also echo the comments that were made from the floor that the D-level PL/I and/or the Version 3 rather than Version 2 compilers under OS would have been interesting alternatives, not necessarily replacements, to what you used. I've never used the D-level compiler; I understand it compiles slower than molasses but that the object code it puts out is pretty good. I heard a presentation once at GUIDE on thoughts of putting the F-level compiler under OS, under DOS in the IBM scheme of things, and someone got up from the floor and said, "Have you thought about putting the D-level compiler under OS? It produces better object code."

I would say "Yes" in response to your conclusion that a nonprofessional would probably be better off using FOTRAN because it's just less complicated for his applications. I would say a hearty "No," however, if that were a nonprofessional who was either intending, or whom his management was intending, to have further exposure in computing. As I said several months ago when we met here, there is a vast body of people in this country, several hundred thousand, unfortunately, who equate the upper bound of computing with what you can do with FORTRAN. That just ain't so. It hasn't been so for a long time. I deplore any constraint on the exposure of programming concepts and programming language tools by, for example, presenting courses in universities that are based on FORTRAN.

With respect to the compiler, your figures are right; there's no question of it, although the Version 3 compiler may perform better. However, I think it's very important to point out the following; as a matter of fact, this is an implicit answer to one of your questions folllowing up. You asked: "What can the industry expect in PL/I implementation improvements?" Well, there are two answers to that question. The first is that we have indeed learned an awful lot more about compiler writing since FORTRAN first came out in 1957, and <u>some</u> of what we've learned is inherent in the current PL/I implementation. On the other hand, I suspect that the current, the initial PL/I

implementation is far superior to most 1963 versions of FORTRAN, for example, and those 1963 versions of FORTRAN were six years after Day One of FORTRAN implementation and FORTRAN thinking. It's clear to me that people will indeed learn more about implementing better compilers for this kind of language. I don't know how much better; I can't give you a figure, but it's very difficult to compare the first generation of PL/I compilers, for example, or ALGOL 68 compilers—you pick it—to the F generation, for example, of FORTRAN and COBOL processors. I'm not saying you shouldn't do it, and I'm not saying you have an alternative; I'm just saying it's very difficult to do that and to draw conclusions that will be valid over a long period of time. It seems to me that one can liken your study—and again, I'm not saying you shouldn't have done it and you shouldn't publish it and promulgate it—but one can liken it to trying to measure the effectiveness of German versus French by determining which one is more easily learned, which one takes longer to learn. It's not clear that you can; I think a reasonable French author like de Maupassant would certainly find French a more flexible tool and more easily learned, but Goethe would disagree to the hilt, and then there'd be those scholars of comparative literature, comparative language, and comparative linguistics who would be sitting on the fence.

Henderson: Mary, do you want to comment?

Lasky: I'd like to know the answers to Bob's questions before we go on, about the educational tools you people were using and so forth.

Henderson: The environment that we operate in is such that we first of all employ above-average professional programmers, and we have never engaged in formally training them. We don't have that big a staff, for one thing, and most of our training is done in a very informal environment. This is true in regard to this study in connection with JOVIAL, in connection with COBOL, in connection with PL/I. Our organization as a whole already had a great deal of experience in FORTRAN, so the problem of picking up any additional knowledge of FORTRAN over coffee was not very difficult. The documentation was a problem, and not only

for PL/I. We fought a lot of battles trying to get documentation, but the timing of the project didn't allow us the luxury of having everything beforehand and then giving a course to explain it. A lot of our learning was done by brute-force digging.

I have to agree that link-edit time has to be factored out of the quantitative data if you're really going to evaluate only the language as such. Obviously we collected a lot of implementation-dependent data which we chose to display today.

I agree with Bob's statement about exposure to a language: what a nonprofessional programmer might want to do or might be called upon to do in the future certainly influences whether an organization would train him in PL/I or in some specialized language.

We haven't factored in anything about the value of PL/I as far as replacing two or three languages that might now be used by an organization. I know only too well the communication and other problems when dealing with a lot of different implementations, using different languages. These haven't been factored into this study whatsoever. The total user cost, as brought out by Mr. Althoff this morning, is certainly a consideration; being able to deal with a single language and communicating across all your organization is an advantage, but each organization would find the relative value different.

Again, we conducted a study in a very specific environment, which we stated. Bob also raised the point about the breakdown of the different kinds of statements. We are itemizing statements and cataloging them in our written report.

Lasky: Coming from a scientific organization, I think that in some ways that you can give a group of highly professional programmers the manuals and they can produce excellent PL/I code. We certainly have a group of programmers that we consider quite outstanding in the Satellite Division at the Laboratory. They have basically taken the manuals and have gone off and in a very short period of time were writing extremely good PL/I code, using controlled storage to allocate their arrays for various things that they needed, using the compile-time features. This is a group of programmers that formerly

could not do their programming except in assembly language. They have never had the luxury of using compiler languages before because they weren't able to do their problems in them. Learning PL/I was certainly very simple for this group. On the other hand, we don't send our programmers off without at least some formal education, ranging from the two-hour course on fancy things in PL/I that I gave to this group to a six-week course which most professional programmers that have been experienced in, say, FORTRAN, are taking. I think that, overall, we have found in the Laboratory that PL/I is extremely easy to learn. People can in a very short period of time be writing good PL/I programs.

The PL/I debugging time we have experienced has been minimal. We have found that when our programmers would write a program in FORTRAN they'd have to have many debugging runs because (maybe it's just the FORTRAN compiler we happened to be running under) the compiler catches things in various phases. A programmer would catch a certain set of errors and then have to resubmit it and catch more errors in a later run. But in PL/I, while the diagnostics that come out may be verbose, the programmer can usually catch all the errors the very first time around.

As for some of these things like leaving off semicolons: maybe they do it at the beginning but they soon learn. Experience is the thing, and when a programmer goes back to writing in FORTRAN he wants to put semicolons at the end of all his statements because a statement just isn't a statement any more unless there is an ending punctuation for it.

As for the nonprofessional scientific programmer: we have found that people in the Research Division occasionally have to write a program when they're on very tight budget. They're not in any respect programmers, but they've done some programming in FORTRAN. They come in and learn PL/I, and all we have to do is tell them that they can use list I/O and don't have to format their input or format their output or use data-directed I/O. Immediately their response to the PL/I language is very good in that it's very easy to learn; it's been our experience that the nonprofessional seems to go to it much more readily than he has to FORTRAN.

Henderson: If we, if the industry now had PL/I subset-directed documentation, I believe it would facilitate the learning process. Perhaps that will come in time, but as far as learning the language right now goes, you get everything or perhaps nothing. There is some problem in wading through it all.

Rubey: I think it should be brought out that a language really exists only in its documentation. There's no sense in implementing a compiler for something that isn't documented. To some extent a language is always going to have to be evaluated against what its standard in documentation is. If PL/I is going to have bad documentation for a long period of time, then it's going to be poorly used for a long period of time.

Rosin: You didn't get my point earlier. I said that the reference documents are not easily used as tutorials, and I'll say that again, but there are at least three publications put out by IBM as red-covered student text which are not horrendous to go through: one directed toward commercial applications, one toward previous FORTRAN users, and one that's called a primer—it's not clear exactly what it's directed towards. And there are some books on the market, including one co-authored by Mary.

Rubey: Right. I didn't mean to imply that our programmers were given only the reference manual for the language. They were given a great deal of literature—a library of literature. I think I probably over-emphasized the lack of support that was given to the programmers. This was a lack of support in the area of the specific problem. They were aided in understanding the language, and on a one-for-one basis. They weren't sitting in a class; there was somebody really helping them to understand the language, pointing out to them good manuals, answering questions they'd bring up. But there was no one trying to sit down and say, "This is a good feature of the language and if you use this, your program's going to be better," because this would have been unfair. We'd have had one person doing the benchmark problems instead of seven people; the programmers would just have been intermediaries between that person and the final result. So it had to be seven people with a wide range of experience.

Rosin: But that's what the programmer was getting over coffee if he was a FORTRAN user.

Henderson: Well, he was getting a certain amount of that over coffee as the study progressed. All of the benchmark problems did not start in parallel, and consequently our organizational experience picked up as the study went along. There was certainly better understanding as the problems were completed and certainly more informal information about the languages available to the participants.

Rosin: I'll say again: I don't object to anything that was done and I don't object to the presentation. I just hope you will footnote it by saying just what you said. I think it would be great if somehow somebody like the Air Force had a few more dollars, which they don't this year, to say, "Switch roles and do it again," that is, let you run a more well controlled and more thorough study if you really want to get results that were meaningful.

Henderson: We were asked to do a lot, but with few benchmark problems.

Althoff: Toward that end, wouldn't it be better to try and get a sample which is more representative of that which the Air Force has to deal with if it's the Air Force that's going to do this?

Henderson: You're talking about Air Force personnel now, or are you talking about Air Force problems, or both?

Althoff: It's the personnel the Air Force will be using to do their work.

Henderson: Well, perhaps it would be a very good experiment for the Air Force to conduct with its own people.

Question from audience: I would be very interested in knowing, based on these limitations and a recognition of the many circumstances of small sample size and other factors, what possible conclusions can come of the study. What can you possibly conclude in consideration of all the factors?

Rubey: I think people tend to be commenting on the quantitative data, and really we tried to make allowances for small sample size and so forth in our answers to the subjective questions. As far as the subjective questions go, PL/I does very

well against COBOL, FORTRAN, and JOVIAL. If it were important for somebody to standardize on one language, and if an efficient compiler were available, then it would be very possible to do that using PL/I. I don't feel that our presentation ever attacks PL/I; I think it to some extent supports PL/I, given a good implementation of PL/I and good documentation.

Henderson: Yes, Russ [Russ Meier, head of Air Force Project 6917, which sponsored the study]?

Meier: Vi, it might be important to mention that we never did envision your contract as coming out with final conclusions saying that PL/I was the greatest thing that's ever been produced. We fought a long time just to be able to get the contract under way to do exactly what you did. Bob's comments are appropriate in that it could be a lot different if we did it again now. The compiler issue is a trap; we aren't interested in the compiler. We were interested in a gross evaluation of language peculiarities, language capabilities, language possibilities, but not in coming up with a conclusion that the Air Force next year should standardize on PL/II or III. I think some of these questions are trying to get you to say a little more than the study was ever intended to find out.

Henderson: I'd like to add on that. We're in no position to recommend to any organization to adopt PL/I or anything else. There are too many things that one has to look at. In fact, one of the general questions I have to address to the panel, if we get to that point, is: "Just how does an organization that hasn't been exposed to PL/I or perhaps is first getting its own computer—how does it intelligently approach this problem of whether it should adopt a language such as PL/I or not?" I don't think there's a simple "Yes" or "No" answer at this point in time. You have to look at the people, you have to look at the computer they're going to get, and so on and so on.

Rubey: We're presenting all the raw data in our final report to let people draw their own conclusions if they want to. I totaled it up—there are some 700 different at-least-one-sentence-or-longer answers to various questions, the result of about 11 people looking at PL/I. So there's a lot of data that people could study to draw their own conclusions.

Henderson: Let's go on to the other participants so we get a complete cross-section. Mike?

Sykes: There's a thing I omitted to mention this morning on the subject of teaching. I think one thing that's very important is that people should learn initially the subset which is likely to be what they need to use. If you're thinking in terms of a commercial-type programmer, you can teach him a subset in which he can do pretty much the same things as he can in COBOL. He can probably get programs of about the same degree of efficiency as he can get in COBOL, but it's an easier language for him to use. We haven't had as much experience as I would have liked to have had in teaching PL/I. What we have done is this: We run what we call a Directors' Computer Course ("director" is English for "vice president"), and this lasts three days. The second and third days are devoted mainly to what things can be done and are being done with computers. But on the first day we start off by telling them what a computer is, and before they dine that evening they get their program working, which is quite a lot to do in one day. It's true!

Henderson: Is that how you evaluate vice presidents?

Sykes: Now, the first dozen or so courses that we had were given in FORTRAN. In fact, we gave them a two-part problem. They had to do the first part on the first day and on the morning of the second day they did the second part. We had to teach them a little bit more so they could get through it. The language we teach them now is a subset of PL/I; we do get on one side of one piece of paper all the syntax they need to know. We teach that in the space of about one hour. This works. We think this is the sensible thing to do. We don't tell them anything about record I/O; we don't tell them anything about data structures. It is basically a FORTRAN subset, but it's simpler than a FORTRAN subset would have to be for the simple reason that we don't have to tell them anything about the rather messy FORTRAN I/O statements with their formats; they don't need to know about formats. So, with PL/I it's very much easier than it used to be. Now they're not going to go away and program, so it doesn't much matter if the programs they write are horribly inefficient.

I participated in another course where we took some research chemists and one or two physicists from one of our research centers. This also was a three-day course but we could teach them a little bit more of the language. We taught them iterative DO's and that sort of thing, but we didn't tell them anything about format. They were writing programs on the first day; they were debugging programs on the second and third days; and they each got several programs working. I think they went away happy with it. This was based on a 14-page document that is more or less a self-tuition thing; in other words, they could pretty well have gone through it themselves. It had examples in it, and also exercises with answers in the back. I believe it was produced by IBM-Australia. It could do with a bit of tidying up, but it is a very handy document to introduce someone to PL/I. Using this, they didn't get into too much trouble. So for them, for what little they know, PL/I was easy to learn, easy to use, easy to debug, and they could go away and learn more about the language if they wanted to. So, for small programs, I don't really see that PL/I can be any more difficult to use than FORTRAN. Unless you take somebody who's up to his ears in FORTRAN, I just can't think it any other way.

Question: A general question: How much trouble do you imagine they might get into with the possibility on the default attributes by teaching a small subset which is not implemented?

Sykes: Very little, I believe.

Question: Do you teach them specifically, for data declarations, for instance, that they have to be very precise?

Sykes: Well, in fact, we told them that the DECLARE statement was necessary if they wanted to use arrays. Otherwise they would have a FORTRAN situation of having floating A-to-H and N-to-Z and binary integer otherwise.

Question: Binary integer? Did they get into trouble on that one?

Sykes: I don't think so, I don't think so.

Question: May I offer a comment on teaching? I've done a bit of instruction in simplified FORTRAN for open-shop users, some 500 engineers at Westinghouse. My simplified FORTRAN was ignoring the fact that there were DO statements, ignoring the fact that there were such things as integer variables; and teaching floating point and doing I/O with preset FORMAT statements, so that the user was not concerned with generating FORMAT statements. The course took three contact hours. Our aim was to make these engineers able to go and program their own work themselves. At Harvard, I expanded the course to five contact hours, and included FORMAT statements, DO statements, subscripted variables, and so on, and made use of the recorded lecture series that Hume at Toronto put out. I would recommend them very highly as a teaching technique. But when I approached PL/I, I thought I would use this same idea of simplifying the language and using a subset, and I thought I'd just forget about the fact that PL/I fixed-point variables were not integers. I found I couldn't. The default options caught up with me, and the programmer who didn't know about these things was hurt by it. Another thing was the default options with respect to the range over which variables are defined; we were always getting caught up in this sort of thing. If you used "X" in one place in a program it meant one thing; if you used it somewhere else it may or may not have meant the same thing or something different. This sort of thing made me wary of applying PL/I to the open-shop user in the same sense that I had been accustomed to doing in FORTRAN.

Rubey: We experienced approximately the same problem through the default options in our evaluation in that they were different in different places. The programmers generally found that the default options weren't immediately useful without knowing what they were going to be. You couldn't just ignore, not know about it. You had to say, "Well, OK, this is a default option; then I don't have to write it down." You couldn't say, "then I don't have to know what it is."

Rosin: I wanted to ask Corby a question in this regard. With respect to defaults, is the implicit statement implemented in EPL? Is there an implicit statement or a default statement?

Corbató: I'm not sure I understand exactly what you mean. There are default attributes.

Rosin: In the original NPL report, an implicit statement was specified that allowed the user or installation to essentially manipulate the default attributes.

Corbató: I consider that a rather small issue at this point.

I think we can draw some conclusions out of the study that's been reported on today—not black and white, but it tells me one thing right away—there isn't an overwhelming difference: we're talking about something less than a factor of two most of the time, and for a rather noisy, semicontrolled experiment with small samples. The art of management is making decisions on the basis of incomplete information; this is what you have to do here. But this inconclusiveness already says, "Well, this isn't the problem to worry about," in the sense that there are other issues poking up through the study like the tips of icebergs, telling you that there are other issues that are just about as important. For instance, it is argued that it's not appropriate to include the link-edit time, but of course it is significant if you're trying to use the language on the particular machine. If you're trying to use it for student use, where you have lots of small compilations, link-edit time is a tremendous millstone which you can't ignore. It has, in fact, discouraged the use of PL/I, even though there's a strong desire to use the language for some of the reasons that have been expressed today. I think it's clear, even with a slight comparison between machines, that there are differences in implementation. It's also clear that this implementation, which is almost exclusively an IBM one, is in its early stages, and I think it can only get better. Although one has to evaluate where it is, I don't consider it the end of the line.

I think there is a learning period which was not well controlled—it's hard to do so with the resources of the experiment—but I think it can only bias the results against PL/I. The semicolon idea would have just vanished if you'd used as your sample of programmers people who had dealt only with ALGOL. There wouldn't be a problem; they would be putting semicolons after every statement routinely.

Question: Even in FORTRAN?

Corbató: Even in FORTRAN. I think it's important to notice a couple of things, though. One, the evaluation is biased to a single point of view, namely, that of the programmer or the effectiveness of programming. That's only a single level. There are other levels from which you could be looking at the language. One of them is that of the manager of an installation; he's worried about the continuity over a five- or ten-year period. He may also be worried about his ability to maneuver in the marketplace. At the moment if he adopts PL/I exclusively, he's also made a complete embrace with IBM. That may not be bad, but it's certainly a decision along with the adoption of PL/I. So one of his criteria might be that he would accept PL/I if he could be assured that there was a version of it with which he could be self-sufficient, which would allow him to bootstrap onto any manufacturer's hardware by his own gyrations and not by begging the manufacturer to supply it with the machine. If that's the criterion of evaluating PL/I, of course PL/I has failed right now—most languages fail. I think that's a rather serious problem, not because I think pro or con with IBM, but because there's a kind of rigidity in the situation which forces people to get hysterical about the choice of languages rather than arguing about their technical merits; and most languages have some pluses and minuses.

Another point of view which has not been discussed at all is the evaluation of the language from the point of view of trying to implement it. Whether you pay for it directly through a vendor or try to bludgeon the manufacturer into supplying it and pressure him to make changes in your own interest and so forth—you're paying for it in the end. It's a major issue; it's an overhead that goes with the language all through its useful life. That hasn't been discussed. I'm not criticizing the study, but I think it is important to recognize that it's only looking at a sector of the problem.

Finally, as a person who's interested in the educational process, I really can't be altogether charitable to the language. I like it in many ways. It's a practical language; I think it's

quite useful to have it here and now. But from the point of view of teaching it, I think it's in bad shape. This may improve as people find slicker ways of trying to sweeten the pill, as has been done in FORTRAN, for example, even though FORTRAN still has a lot of ugly glitches. PL/I has not been helped by the attitude of some of the original people associated with the design of the language; they made in one article a literal boast that the language lacked formal description, which can only lead to chaos in trying to determine the specific intent in design of PL/I. And it has; in fact, there are telephone directories of design clarification which are grotesque for anybody faced with trying to implement it. This kind of know-nothingism, which was part of the original spirt of the language and has been to some extent corrected, still has a price attached to it. Namely, it's hard to teach the language in the sense that it's a big stew-pot; it's hard to break it up into packages of ideas which may be taught in sections so that a person learns a little bit at first and then learns a little bit more. I'm not convinced that one can't salvage the situation by some shrewd manual-writing wherein one devises subsets of ideas which are of increasing usefulness or interest, and I hope that happens. But I think it's rather difficult at this stage of the game to salvage the implementation ease by writing the language such that the layering extends into the implementation—so that the implementation is also broken down into a hard core and a layer of extra goodies and so forth. That will probably require another version of the language, or a successor, which will, I'm sure, come in time and will be necessary for the evolution of the computing field. I think one has to be a little relaxed. PL/I is here today; it's a useful thing. On the other hand, I don't consider it the end of the line.

Henderson: I'd like to reinforce the fact that our evaluation was indeed programmer oriented, and that there are certainly a lot of other factors to consider from a management point of view and from a cost point of view. I think perhaps Earl would share that opinion.

Althoff: Yes, I did give a few of those management-type evaluations in my speech. Generally speaking, the conclusion I had in it was that the case for PL/I was pretty clear if you had a situation where you were starting a new installation with programmers

new to higher languages, you were going to a new machine, and you felt that you would stick with IBM for the next four or five years. Where an installation has about five or six different languages and they seem to be meeting its urgent needs and it has a small number of programmers, I don't think the case is too clear. In some cases the question whether you can stick with one manufacturer for four or five years does raise its head—not too frequently, but in some of our units it does. We get into all of these factors, and that's one reason I showed you the chart on selection technique. We found so many factors we could sit around and discuss them for a week and come to no conclusion.

I did have a few comments I wanted to make about the presentation. In particular, there was a conclusion that said that COBOL was preferred for the nonprofessional programmer. Now, we don't have a single nonprofessional programmer that will agree to use COBOL, either commercial or scientific, so this statement does leave me a little perplexed. We do have quite a few that are using PL/I. And I found one of the advantages in our company was that PL/I would be used and is being used by accountants, and some of what I called the statistical people who are involved with the commercial people—so at least if I were the analyst I would have picked PL/I there.

The second comment I had was on the first benchmark problem, the one concerning payrolls. We have to write something like 300 to 400 programs of this type per year. It did concern me approximately a year ago that people writing those programs felt that, for some reason, PL/I was pretty difficult in comparison to the other languages. We got a study group of a couple of people to look at PL/I and try to pull from all the spots in all the manuals how to write reports. This actually resulted in a chapter which has been contributed to GUIDE. They picked the array handling—something that everybody would swear was scientific, so the commercial people hadn't bothered studying it—for automatic totaling and described how it could be done. I think that the conclusion of our study group was that PL/I is superior for that type of program.

Another comment I'd like to make, both from the professional view and the company view, relates to missing items. When-

ever we're at GUIDE or SHARE meetings we keep standing up and urging IBM to get these things in the language. However, from a company standpoint we really haven't really found them to be major stumbling blocks. Sort has come up, for example. Professionally we're working on three different possible sorts that could be called by PL/I: internal sort, a kind of external sort within the program, and what I'm calling the operating system sort. But as a company we simply have a linkage and we call sorts for PL/I programs all the time. So we don't visualize this as a huge problem from our company viewpoint. We've been working on data validation professionally, the problem mentioned by Mr. Sykes. In fact, the PL/I project group from GUIDE is meeting with IBM this week on trying to resolve better ways of doing this. The major problem we've run into is that a lot of the people feel that all of the implementations proposed destroy some of the machine independence and we're trying to find some way that leaves things to the machine. How can you have a zone on a tab card where Column 79 tells you what the assignment field is in Column 12 and call that machine independent? However, from the company view, we really don't have a problem. We have an 800-position translate test type assembler subroutine that can be called easily linked with a PL/I program, which we just go right ahead and write.

I think much of the same is true of the diagnostic problem and the assembler problems you had in the study: you've got to keep the two views separate. In my estimation, those are the main missing items: sort, data validation, and perhaps some improvement in catching the too lengthy comments that don't stop with an asterisk and a slash—but we have standards and coding forms to stop these, or at least catch them. On the whole, I think the presentation pretty much gave the same conclusions we were giving. PL/I is the preferable choice of the separate languages.

Rosin: With respect to Corby's comment on the link editor: yes, you do pay that price if you use PL/I on your OS now. I agree, except that some of us have been exploring the possibility of writing a program whose name is L-O-A-D-E-R, and a few of these programs do exist, and, as a matter of fact, they promise to L-O-A-D PL/I programs. We suspect link-edit

time will go down under OS; we're not sure, but we suspect it will.

Your comment on the lack of formal definition is appropriately painful. One thing that's happening right now—and I'm sure it's well known—is that IBM is developing a formal definition for PL/I: both syntactic and attempting a semantic definition by defining the PL/I machine in a sense against which PL/I works. The analyzed PL/I programs are interpreted. One thing that they must be coming out with—and I'm sure that there would be a consensus on this—is that the language could be a lot smoother. The fact that the array expressions aren't array expressions becomes painfully obvious when you have to write it out in BNF or modified BNF, et cetera, throughout the language. There's no question of it. The only painful thing is that I'm afraid that eventually there will be a successor to PL/I, not in the sense that PL/I will be assassinated and this new thing will move into its throne, but that eventually this other thing must come along.

Henderson: Let me address the panel now with a question. I think that there is a great deal of the industry that does have questions on its mind whether or not to undertake programming efforts using PL/I. There are many factors, I'm sure, which occurred to you today which influence such a decision. I would like to ask the panel: What criteria should an organization really use in making a decision whether to adopt PL/I or not? We can take the range of organizations from the Air Force in toto to that organization which doesn't even have a computer today and possibly will be getting one in the door in another three months. Just what should their approach to this problem be?

Sykes: I think I would agree with Mr. Althoff that anyone who's getting their first computer—if it's a computer such as one of the existing series of the Model 360 and it's large enough to use sensibily one of the existing PL/I compilers—I think their only decision from my part of the world is whether they would use PL/I or would stick to assembler language. I don't think they would consider using COBOL or FORTRAN unless there were

special circumstances. For example, Rolls-Royce (they are not, of course, coming to computing for the first time; they've been in it for some years) do have to use FORTRAN for writing certain programs that they have to supply as part of the package that they are selling—for simulating aero engine performance. They use FORTRAN because their customers tend to have machines with no PL/I compilers. I think this is the main consideration in U.K. as to whether you adopt PL/I as your high-level language or not. Rolls-Royce use PL/I very extensively; they have something like 200 programmers using it, and they only resort to FORTRAN in this sort of case. For a large organization such as ours, I think it depends as much as anything on how interchangeable you want your programmers to be. We do want to be able to move programmers from one project to another and we don't want them to have to learn a new language in the process. We think this is quite important, particularly as we don't see much advantage in going to either COBOL or FORTRAN for any specific purposes.

Henderson: But your ground rules are based on the fact that you have a computer system for which the PL/I language has been implemented. Would you go so far as to say that the selection of a computer these days should be heavily dependent upon the availability of a PL/I compiler?

Sykes: I certainly think there are sufficient advantages in having PL/I available, that in selecting a computer one would certainly want to take this into consideration. I doubt whether it could really be made a dominating criterion, but I certainly don't think I'd leave it out altogether.

Lasky: We chose to go to PL/I as the major programming language we'll use on our 360 Model 91. We did this for the main reason that most of our programs at the present time are written in assembly language on the 7094. We do not want to be tied to a particular computer again. We find that the transition of going from one computer to another computer is extremely difficult if everything is in assembly language, and as we go on in our future plans it will become much more so. FORTRAN

just will not satisfy the requirements of what we need to accomplish; and so if we kept with FORTRAN on the 360, we would have to resort to assembly language, which is what we want to get away from. We find that PL/I answers, except for some of the very critical timings, all of our problems. The programming is very easy to do in PL/I, and we find that we don't have to resort to assembly language.

Another major factor in going to PL/I was that we were able to write a converter/translator from FORTRAN to PL/I. We wrote this in PL/I, so it helps with our transition of getting from one language to another. I know Mr. Althoff brought this up this morning as one of the considerations of looking at a language, and I think it's one of the very reasons that there have been so very few major programming languages so far: economic considerations inhibit the transition from one language to another.

Henderson: Was your decision to go to a 91 instead of, say, the CDC 6600, possibly dictated by the fact that PL/I compiler would be available for it?

Lasky: Well, part of our discussions with CDC centered around the fact that if their equipment were chosen, we would want the capability provided in a high-level language that was the kind of thing that would be provided in PL/I.

Rubey: As of this date, though, you don't really have machine independence because you're locked in with IBM. Suppose you wanted to go to a 6600 a year from now, or you wanted to go to a GE 635. So have you bought any machine independence by this movement?

Lasky: We are definitely out on a limb in that respect.

Corbató: You bought IBM machine independence!

Henderson: Bob, do you have any comments?

Rosin: I can say the following thing about your question. It forces more questions, for example: What applications do you have in mind? How much in PL/I do you want to have available?

Do you have old programs to convert? What machine are you talking about? How big? What kind? What specs, in general?

I would say that I wouldn't let PL/I frankly dominate my choice of a machine, but I'll add the following comment from a pedagogical point of view. We're offering a course this semester called Programming Languages in which we're trying to cover language as well as implementation. Of course, it's impossible in one semester, but PL/I as an example raises more interesting questions about generality, about facility and function than any other single language that you can talk about. There are all sorts of questions about implementation.

In regard to Mr. Althoff's comments, let me raise one other point which is very indicative of what's happening in this world. I think we should all be aware of it. When the 360 first came out, IBM said, in effect, "You will use the following languages we'll make available, including PL/I, and thou shalt not touch OS because thou shalt have thy fingers burned," and we, in general, I think to a man, to an installation, followed their indications. When PL/I first came out it was understood that it was absolutely impossible for any mortal on earth who had not implemented a PL/I compiler to write an assembly language routine that would link successfully with PL/I. Now it turns out that kind of talk just isn't true any more, and an installation like Mr. Althoff's is indeed writing assembly language programs conveniently, effectively, and efficiently to link with PL/I, just as we did for years with FORTRAN. We knew it wasn't clean, but it got the job done. I would submit that anything that Mr. Althoff wants to do in PL/I—for example, sort, RPG, a facility for deciding whether or not a certain field is numeric, something removing overpunches—can be programmed in PL/I. I also agree that he does not want to pay the price at times, nor do I, and therefore we escape to another language, at least for the short term, to solve that problem. But we're not afraid of the machine any more, or the system.

Althoff: I think the first thing you must try to do in deciding whether to adopt a language is make sure there isn't some overriding thing that makes the decision for you. If, for some reason, your company says there must permanently be five suppliers

that can serve the language, you quit; your decision's made. However, I think if they say, "We're making a choice now for four years," and all they want at the end of it is assurance that somehow we can get out of the "trap" if for some reason we decide initially on some manufacturer that doesn't have the language, then all we have to do is show them our capability to write a translator to the presumed new language—ALGOL 69 or PL/7 or whatever. I think that some of the work that Mary Lasky's done proves that we could do it. That's a cost at the end that we put in our selection. One way of getting out of it is to write a translator. Of course the other way of getting out of it at the end of the four- or five-year period is if the new manufacturer that we are considering has a PL/I that we can go to at a reasonable cost or has 360 code in an emulator. But this is just an example; normally we find out whether there is an overriding criterion that management has set that says "No." If not, we go in and weigh all of these factors. Until this industry settles down and standardizes hardware and everything for a good many years, can you ever get the situation where you don't have such a major conversion hurdle after a period of four years of being tied to some supplier? I know there are companies that say, "We've got to have at least three suppliers," and they think it's perfectly all right to stay three years behind the field.

Corbató: I think the issue is not quite that simple, because if you pick a single supplier you are in a sense becoming a voluntary partner in price fixing, and you've lost all your leverage and your ability to change if a better supplier comes along.

Althoff: I did want to point out we have four computer suppliers.

Corbató: OK, but if we're talking about PL/I, it's only on one machine. My argument vanishes if it's on every machine. I'm not trying to make the case for or against IBM; I'm trying to talk about the problem in the abstract.

Althoff: We have a General Electric process controller tied to a chemical process, and there's no way under the sun we can switch that from General Electric to any other company.

Corbató: I don't think the problem is unique to PL/I. I'm trying to talk about a problem in commitment to a unique system. I think it is a management policy decision, whether they want to temporarily—four years in a sense is temporary—abrogate this privilege of being able to get the best possible cost-effectiveness on an annual basis. At the moment the computing field is so much in a shambles that annual changes, unless they're insulated from the practicing programmers, are just chaos. I also consider the translator and/or emulator to be a bit of a gamble unless one knows the end transformation already; but one doesn't know, that's the future. It's certainly a sensible and savvy way to go, and it's perhaps a gamble, but one considered worth talking.

I would raise a larger issue, and I think it's one that every company or every organization I know is faced with today: There has been a very belated recognition of the need for re-investing in software development and understanding and a general support staff in every organization I know. They're all running on a shoestring in terms of number of system programmers, the number of people that can understand and teach, and they're all on thin ice the minute a system change comes through that isn't quite right—somebody drops a card somewhere and the whole institution may be set back or shaken apart. To some extent, I consider this to be a positive reason for going to PL/I. In other words, one should be investing in capital improvement. I feel the PL/I argument goes down in importance if it can be a choice, and where the user is aware of what kind of a trap he might be getting into and the fact that he may have to convert later by hook or crook to some other machine. Unfortunately—well, I think unfortunately—some companies feel that it is necessary to dictate a single policy to keep more uniformity in the interests of reducing costs. I think it's a little short-sighted; I can say that because I'm not responsible. I may be wrong.

Henderson: Ray, do you have any comments on the last question?

Rubey: It's a very, very complex question. There are a lot of things that go into it other than the language, like the manufacturer. There is also the question of what kinds of inefficiencies you are willing to tolerate. If you're going to buy a machine you

want to know the cycle time, the add time of the machine, the memory access time on the drums and things like that. People pay a lot of attention to the hardware, and then they sort of accept the software given to them by the manufacturers without the solid guarantees generally that they get on the hardware. You'd probably send the machine back if the memory cycle time was 10 times what the manufacturer told you, but if the software takes 10 times longer than you'd expect to roll off the drum, or something like that, you have no guarantees on that matter. So to some extent, you're taking software on faith because it's "given" to you. Your decision whether to go to any particular language depends a lot on how well you believe your supplier.

Henderson: I had some more questions but we're running pretty late. Let's give the panel members an opportunity to address each other and then call it quits.

Rubey: I'd like to go back to some of the comments about our presentation. First of all, there was a lot of talk about the better diagnostics of PL/I. But in one sense they could be improved a lot. I think that many of our programmers will agree that the most common diagnostic message is DATA INTERRUPT. You could spend a lot of time trying to find what the data interrupt means. With any new language, when you start out the diagnostics aren't all the best, so to some extent I think the fact the diagnostics are less refined in PL/I than in many other languages probably contributes to the longer debugging times we got with PL/I.

For the people who were wondering why the nonprofessional business-type programmer might be better off with COBOL than PL/I: there are some things missing in PL/I. For example, sorting is a very common thing that a nonprofessional might want to do. He says, "I've got this; now I'd like to sort it." But you can't give him a series of articles from the ACM Journal about different kinds of sort routines and expect him to pick that up. This should be something that he doesn't have to know about; he just puts it in order.

Althoff: But, isn't that the sort of thing that's coming along?

Rubey: Yes, but in our study we're talking about the language the way it is today and what you can reasonably expect to get with the current implementation, what it will do now.

This refers to subsets too. You can give the nonprofessional a subset or you can give the beginner a subset of a language, but, again, that's something that isn't in PL/I today. There's nothing to hand somebody and say, "OK, you have a business problem; here is what you use." There's no, say, set of default options. You can't give him a subset and say, "You put this card in the beginning and you're going to get the business subset." The default options were a problem in our study. Our business programmers said they were all wrong for them; they didn't like whatever the default options were. For the nonprofessional user, you're first of all going to have to define for him a subset and it's going to have to be arranged so he has the right default options.

Rosin: Let me respond. Those are good points. This sort thing is a particularly good point in the following sense: the facility should be there. I agree with what Corby said this morning, however; that is, I'm not sure that the facility need be in the host language. And this is where Mr. Althoff and the GUIDE committee and I are often at loggerheads. A facility should be in a language for calling a sort as something that may be in the library, perhaps as a library routine. What it's coded in is immaterial. It's not clear that it has to be in the language. The same is true for RPG and a bunch of other things. I think the scientific programming community is a little jaded and a little smug on this point because we have gone with subroutine libraries for years and years. The Michigan Library at Yale right now has about 700 entry points in it, any one of which is callable, any subset callable from a running MAD program just by mentioning their names. We can be very smug about that. This has not been true in the commercial shop to a great extent. I think it will come.

Corbató: Germane to that point is the reason for the pressure to put these features in the language. I think the reason is that too often the language designers have taken the rather arrogant attitude that if it isn't part of the syntax, it doesn't have to be

designed. Now that's the problem. Basically it's a reaction of trying to get somebody to think it through.

Rosin: I agree. We passed a resolution in SHARE, I think about six months ago, saying that some things shouldn't be in the language at all and should be taken out, and other things which should be in the language should be integrated into it.

I thought in response to the point you just made, however, I would give a very brief four-item report on some of the things that came up at the very recent SHARE meeting in regard to what the SHARE project and by implication IBM might be thinking about PL/I language development. The first is facilities for changing implicit and default statements. We agree, for example, there is no reason in the world in another environment, say other than commercial, to say that "By default, unless I say otherwise, everything is a varying character string," which for some applications is very useful. There is no way to say that in the language now. The constraint we've placed on this, as a matter of spirit, is that it should be possible by using these facilities to first remove and then reinstate all of the current default and implicit facilities in the language, including default attributes for constants.

The second item is the ability to monitor and handle "unsolicited" interrupts, those interrupts which I guess were previously referred to as asynchronous, not quite asynchronous but at the same time not quite synchronous. This certainly ties to things like teleprocessing and graphics. The SHARE project would rather see that put in the language than, for example, a whole monstrous set of language that's tied specifically to teleprocessing or specifically to graphics.

The third point that's still under great debate—and this is way, way off in the future—is the concept of expansible compilers. This is coming. Somewhere I know there will be such facilities in processors for languages other than PL/I before PL/I has it, but we at least are thinking about it.

The fourth item that came up, again in terms of generality and smoothness, was array handling.

Rubey: Another thing about default options: if you have a lot of programs to convert, the default options now tie you to a particular implementation of the compiler. Your manufacturer may say, "Boy, I've got the superduper version No. 93; it runs twice as fast." But unless it has all the same default options, you're going to have chaos. We're talking about different ways of changing, say, to suit the business programmer. The default options are going to have to change as the business programmers gain experience. Handling this is going to be a difficult procedure.

Rosin: I'll give you an interesting example with respect to defaults which, as far as I know, is not clearly defined. It says in the language manual, not the implementation manual, that identifiers whose names begin with the letters I through N are fixed binary. The question is: Is that I through N going I-J-K-L-M-N, or is it I through N going I-H-G et cetera? This question comes about because the collating sequence is not defined in the language. That's an imprecise feature of the language. As Corby points out, it results from the fact that there's no precise specification, and this is something that occurred to us in our considerations.

Lasky: Some people seem to feel you must always bank on these default options. Another way of approaching the whole thing is to declare everything explicitly, especially for beginning programmers. When IBM brought out the language they shouted from the chimneytops about all the wonderful defaults. I think that often when people don't use the defaults but go ahead and declare, they stay out of a lot of problems that can come up.

Rubey: I would like to respond to another comment that indicated our study might have been biased because the programmers had a lot of experience, say, with FORTRAN. I think the most startling thing is that we took people who admittedly did have that sort of bias, but look at their responses when we asked them, in effect, "We've got the same problem for you, a little different, but a similar problem. Which language do you prefer?" I think in six out of seven or at least five out of the seven cases they said, "Well, I'll take PL/I." And in the other cases they said, "Well, maybe I'll take PL/I." In

general, in spite of their background, they're pretty solidly behind PL/I.

Henderson: Let's close the session for today. I want to thank each and every member of the panel and the audience. I hope that what's been said here today will help you appreciate a little better what's going on in the industry. I hope that all of you remember the qualifying remarks that we've placed on our study results. Given another set of circumstances, a different environment, and a different time frame, you could very well see a different set of results. This indeed was an experiment; the complete results are presented in our final report.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Logicon, Incorporated 255 West Fifth Street San Pedro, California 90731	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b GROUP N/A

3. REPORT TITLE

PROCEEDINGS OF PL/I SEMINARS, December 5, 1967 and March 5, 1968

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

None

5. AUTHOR(S) (Last name, first name, initial)

None

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
April 1968	189	

8a. CONTRACT OR GRANT NO. F19628-67-C-0396 b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-68-154 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) Logicon Report No. CS-6819-R0106

10. AVAILABILITY/LIMITATION NOTICES

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Command Systems Division, Electronic Systems Division, AFSC, USAF, L G Hanscom Field, Bedford, Mass. 01730

13. ABSTRACT

This report consists of the proceedings of two seminars at which presentations were made by experts who had actively participated in the design of the PL/I language or who are engaged in implementing systems using PL/I. Each presentation was followed by a question-and-answer session in which the audience participated, and each seminar included an informal panel discussion among the authorities present.

DD FORM 1 JAN 64 1473

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
PL/I - Evaluation						
PL/I - Teaching						
PL/I - Scientific						
PL/I - Command and Control						
PL/I - Strings and Arrays						
PL/I - Business						
PL/I - System Programming						
PL/I - Panel Discussions						